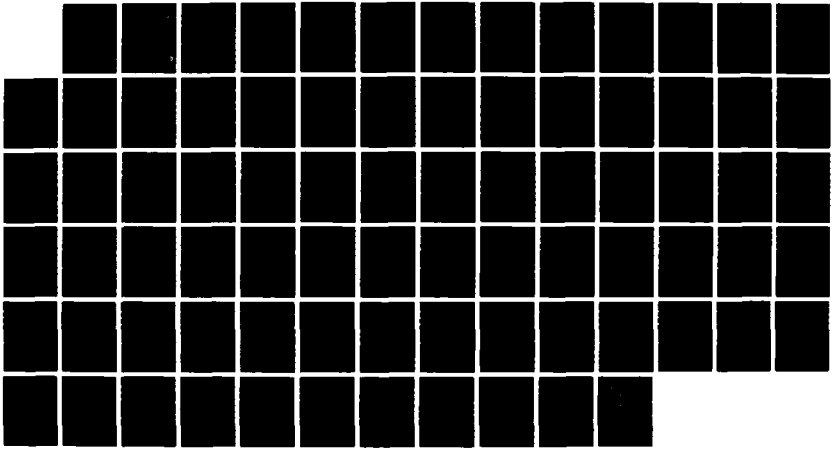
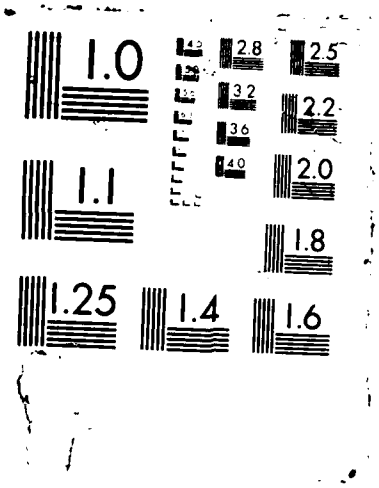
AD-A190 267 AN EXPERT SYSTEM SUPPLEMENTED BY A DATABASE MANAGEMENT SYSTEM SERVING AS A.. (U) WASHINGTON UNIV SEATTLE R LONG 1988 N00228-85-G-3363 1/1

UNCLASSIFIED F/G 10/2 NL




1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0

An Expert System Supplemented by a DataBase Management System Serving as an Electrical Distribution System Engineering Aid

by

ROLAND LONG

A thesis submitted in partial fulfillment
of the requirements for the degree of

Master of Science in
Electrical Engineering

University of Washington

N00228-85-G-3363

1988

Approved by _____
(Chairperson of Supervisory Committe)

_____

_____

Program Authorized
to Offer Degree Electrical Engineering

Date 12/14/87

Master's Thesis

In presenting this thesis in partial fulfillment of the requirements for a Master's degree at the University of Washington, I agree that the Library shall make its copies freely available for inspection. I further agree that extensive copying of this thesis is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U.S. Copyright Law. Any other reproduction for any purposes or by any means shall not be allowed without my written permission.

Signature [signature]

Date 12/14/87

Table of Contents

List of Figures

List of Tables

List of Abbreviations

DBMS	data base management system
ES	expert system
FNF	fourth normal form
SCADA	supervisory control and data acquisition
PI	program interface
RAM	random access memory
LHS	left hand side
RHS	right hand side
AI	artificial intelligence
A&E	architectural and engineering
HVAC	heating, ventilation, and air-conditioning
EMCS	energy monitoring and control system

Introduction

Electrical distribution systems provide power to loads. Numerous individuals are involved in the design, operation, maintenance, and modification of these systems. Traditionally these individuals have relied on drawings, operations manuals, meter readings, and experience as primary resources in accomplishing these tasks. Emerging as a new resource is the Expert System.

Expert Systems (ESs) are knowledge based computer programs which provide assistance in domain specific problem solving. They solve problems by duplicating human problem solving methods using deductive reasoning techniques in conjunction with factual information to analyze a problem and reach a solution. The required reasoning skills can be obtained by analyzing the problem solving methods of human experts. Factual information can be organized as individual fact statements or, in the case of large systems, the information can be gathered and organized into database form by DataBase Management Systems (DBMSs). The codification of these reasoning techniques and the factual information makes up the knowledge base of an ES.

The motivation for investigating electrical distribution system ESs comes from the potential power of the computerized application of reasoning skills to power distribution problems. The present decision to consider ESs can be likened to the need to use computer based electronic spreadsheets such as Lotus 1-2-3. Engineers have gotten by for a long time doing design cost estimates from a take off sheet with an adding machine or calculator. However, with the advent of products such as Lotus 1-2-3, engineers have switched in order to realize both improved speed and accuracy in their work. Similarly, the ES offers both speed and multiple solutions while it minimizes the possibility of overlooking information. ESs also have the advantages of incorporating the best methods to solve problems, being convenient and available (ESs don't take vacation or get sick), and always working their best.

In addition to reasoning abilities, in the electrical distribution setting, the ES must access and deal with large amounts of information. This project supplements the ES with a DBMS. The prominent features of the DBMS (optimized data storage, retrieval, and manipulation) aid the ES in dealing with the large quantities of distribution system information.

This thesis demonstrates an ES concerned with electrical distribution system connectivity problems. Connectivity concerns the paths for transferring power from sources to loads. Using DBMS database information, the ES analyzes how the

distribution system provides loads with power. The ES additionally determines the effects of various system faults, such as line failures and the opening of circuit breakers. The results of the analysis includes determining which loads have lost service and which circuit breakers or lines have been rendered unusable. To restore power to de-energized loads the ES analyzes the remaining system for possible alternative service paths.

In support of the ES a DBMS manages the information necessary to represent the distribution system and simulate modifications to the system. The DBMS program uses the relational model implementing those essential relational features required for this application.

Chapter 1 discusses the motivation behind this thesis project, defines the ES problem area, outlines the joint ES-DBMS solution approach, and provides general background information. Through the use of an example distribution system, this chapter reviews the nature of the work required of those who manage and maintain the distribution system and how they approach their work. This review illustrates the need to develop an engineering aid for analyzing system connectivity problems. The chapter shows that problem solving techniques can be separated from the facts of a particular problem. This separation supports supplementing an ES with a DBMS for problem *information management*. The chapter concludes with background on DBMSs and ESs.

Chapter 2 examines the DBMS developed in this project. The chapter's emphasis is three fold. First, the chapter explains the model used by the DBMS to describe the example distribution system. Second, examples of DBMS operations provide insight into the power of DBMSs as information managers. Third, this chapter discusses program design and coding considerations which result from implementing the ES and DBMS programs in Borland Turbo Prolog on an IBM AT.

Chapter 3 discusses the ES itself. The primary emphasis is on how the ES solves problems concerning the delivery of power to loads. Examples that make use of distribution system excerpts show the nature of power delivery problems and the approach taken in solving them. This chapter also explains the methodology the ES uses to reach solutions by walking through the example problems using pseudo-code rules.

Chapter 4 provides concluding remarks.

Chapter 1: Motivation and Background

Problem Introduction

This investigation involves the development of an Expert System (ES) and a relational DataBase Management System (DBMS) written in Borland Turbo Prolog (Prolog) to run on an IBM AT. The ES is an engineering aid for analyzing the connectivity problems of electrical distribution systems. The DBMS supplements the ES as an information manager.

This study is based on an example distribution system and responds to the needs of those who manage and operate this distribution system. This chapter presents the example distribution system; introduces the individuals who manage and operate this system; and reviews their tasks, resources, and problems. This review establishes that the topical area of connectivity analysis is worthwhile and that an approach using an ES supplemented by a DBMS is appropriate. The remainder of this chapter includes reasons to support the proposal of an approach composed of an ES supplemented with a relational DBMS; a discussion on the general nature of ESs and the Prolog implementation; an introduction to relational DBMSs which includes developing the database for the example distribution system; and a discussion of the relationship between the ES and DBMS.

Example System

The example for this work is the electrical distribution system for the Bangor Submarine Base near Bangor, Washington. The Bangor system is relatively new, less than 10 years old. The Bonneville Power Administration (BPA) serves this distribution system with a single 115 kv feeder. The submarine base distribution system begins with a 115 kv ring which serves six 115/12.5 kv substations. The six substations provide power to the remaining distribution network. In addition to the 115 kv ring feeder, the substations can be fed by both permanent and portable diesel generators. Figure 1 shows a portion of the system.

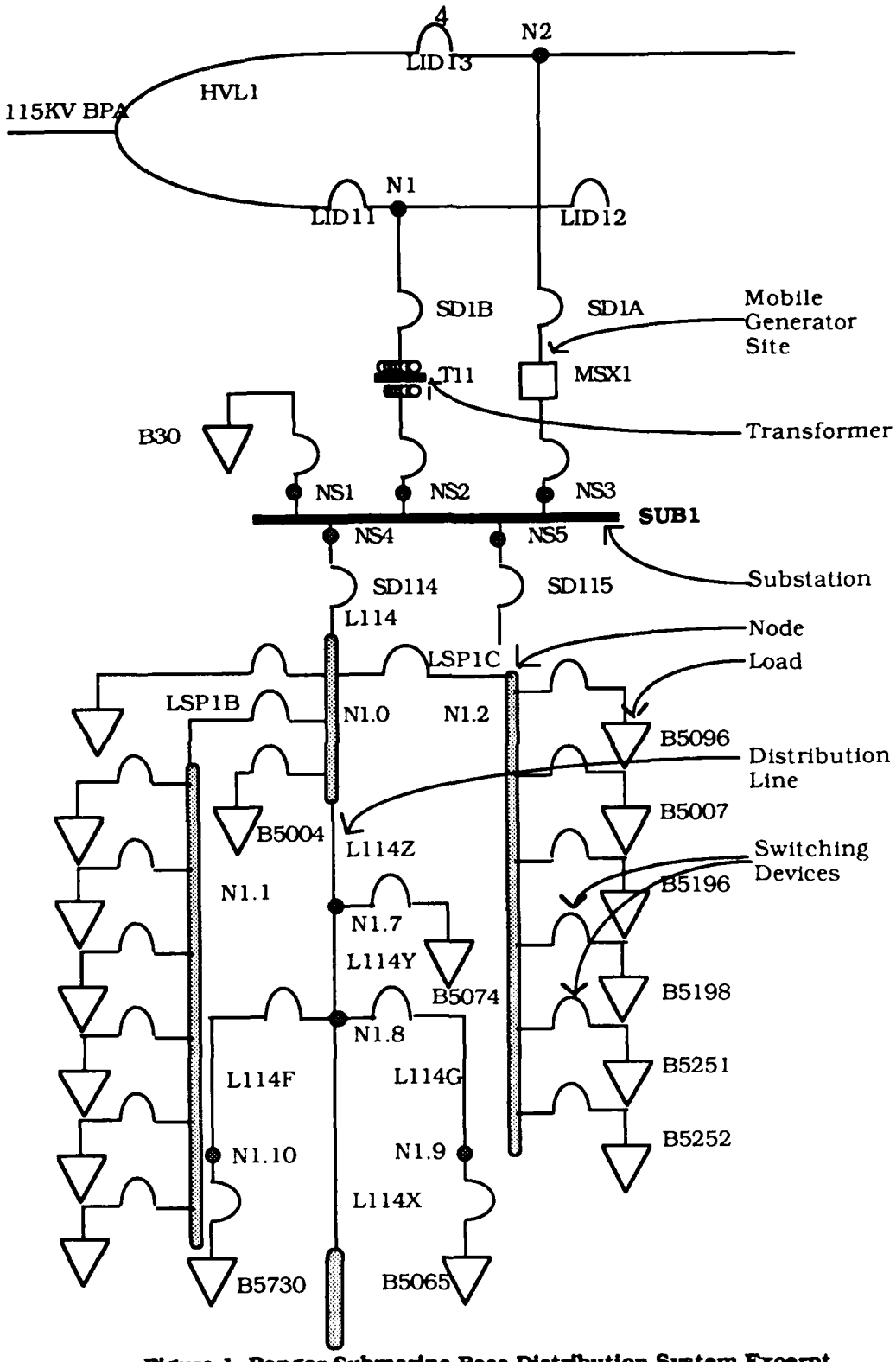


Figure 1. Bangor Submarine Base Distribution System Excerpt

Expert System Users

In determining the specific nature of an electrical distribution system ES one must first answer who will benefit from its development. Answering the question largely means identifying those involved with the operation, maintenance, design, and modification of the Bangor Submarine Base distribution system.

A small staff of military officer-engineers and civil service employees provide overall management of the distribution system. Permanent civil service apprentice and journeyman level electricians accomplish daily operations and routine maintenance. Work of a larger nature (alterations, additions, and complex maintenance) is primarily accomplished through the award of competitively bid contracts. With the exception of only the largest contracts, this work is restricted to small business contractors. Both staff electricians and contractors perform emergency work. Private architectural and engineering (A&E) firms provide engineering designs. Civil service staff engineers are charged with reviewing this design work. Management of construction and maintenance contracts falls on another office. Construction contracts are managed by military officer-engineers and civil service engineers. Inspectors perform daily quality assurance of contractor work and assist in coordinating utility outages. Outside A&E firms may supplement their efforts.

These groups are potential ES users and represent the current human experts.

Expert System Domain

The broad group of individuals needing to understand the operation of the distribution system are the target ES users. Recent research into electrical power system engineering aids includes specific areas such as alarm and event message processing, isolation of faulted line sections[1,2], reactive power and voltage management[3], and operator training[4]. The problems of those using the Bangor distribution system generally focus on understanding basic system operation and possible alternative system operations. Specifically, how does the system now provide power to loads, will opening a specific breaker or taking a line out of service adversely impact service to any load, and what are the possible recovery configurations for various fault situations. As distribution systems are generally over designed, system connectivity--providing a path from a source to the load--rather than system loading is the primary consideration. The issue of system loading and loss minimization through

efficient loading has been addressed in previous research[5]. Reviewing the tasks and resources of those responsible for the distribution system will show that analyzing the paths for providing power from sources to loads is a worthwhile ES domain.

Managers exercise oversight control of the electrical distribution system. Their concern is that the system reliably provides loads with required power. Management may be made up of architects or engineers of any discipline. They typically do not personally seek out detailed technical information. They do, however, need a good overall understanding of electrical distribution system operations including the effects of various actions on the system. For instance, managers would need to know if power could be provided to an operations building if a given substation or line was taken out of service. For these more general questions, they probably already understand how a distribution system generally works or can familiarize themselves with distribution system principles from an elementary text. For how their system works and the outcome of a specific question they additionally refer to one line system drawings and system operation manuals.

Staff design engineers and private A&E firms carry out needed project studies or designs. A design of any size will involve a project manager and a design team. The project manager, like management, may be an architect or any type of engineer. Project managers also primarily concern themselves with overall operational issues. They need to understand the 'big picture' in order to give direction to and provide design team co-ordination. A project manager augments his professional experience and design concepts with project specific information. The design team turns general concepts and approaches into project plans and specifications. They too begin with their own methods and approaches for doing a particular type design. If they don't have an adequate background for a particular project they find reference materials--design guides, code books, and handbooks--which provide them with the 'how it is done' methods. To begin the design they also need a full and complete knowledge of the project system. The design team, using their experience and project specific information, develops design alternatives. Each alternative must be tested to ensure it works smoothly with the rest of the existing system. Testing may involve amending a set of system drawings to reflect the results of the proposed work. The 'new' system can then be checked to verify its operation under normal and fault conditions. This step must be exhaustive to ensure nothing is overlooked. Considerable design time is devoted to getting up to speed with the project system and testing design alternatives.

Permanent system operators are the most familiar with the distribution system's operation. To understand the nature of this experience consider the training

of a new operator. The new operator is first briefed by an experienced foreman. The foreman, using a one line drawing, provides a system overview. He discusses general system configurations, system capacities, equipment ratings and types. The new employee next receives a field tour. Emphasis is placed on major facilities (substations, diesel generators, and large loads) as well as key or troublesome components. While training, the operator becomes familiar with system operations and parameters through detailed system and component drawings and their manuals. The more experienced operators supplement the training with explanations of how 'things are done here.' Eventually the operator has seen most of the equipment and system operations several times and begins to know how this distribution system works.

Distribution system operation generally does not require frequent changes in the way loads are served. When alternative service paths are required, the design usually provides several options. With time, operators learn the best or easiest means of serving each load. Operators rely on experience and seldom need to make more than slight use of reference materials. On a day to day basis experience proves quite satisfactory. Even if a fault occurs on the system, the operator can refer to Supervisory Control and Data Acquisition (SCADA) system output, meter indicators, and field inspection to determine the system's faulted condition. Since the system is operated in a limited number of configurations, the operator generally has no difficulty visualizing a corrective configuration.

Unfortunately, when the fault is in an unusual place the operator may stumble. If the system is in an unusual configuration to begin with, the operator's experience may not offer a solution. Multiple faults can also undermine reliance on experience. At times such as these the operator returns to the drawings for specific system information to find power paths for de-energized loads. Since distribution lines are normally oversized, line loading is a secondary concern to finding an available path.

Maintenance and construction contracts often require the distribution system to be placed in unusual configurations in order to accomplish desired work. At these times the contract managers, inspectors, and operators need to satisfy themselves that the procedures suggested by the contractor or outlined in the contract will work without causing a power loss. The proposed operations are analyzed being very careful to ensure all affects are considered. As contractors, inspectors, and contract managers tend to be the least familiar with the system, the necessary analysis is a time intensive effort.

While each of these groups have different motivations for understanding the basic operation of the distribution system, they share the need to be able to accurately analyze how the system works. This common need includes a general understanding of

how the system serves loads, knowing what results from various system operations, and the need to restore de-energized loads in a timely manner. Each person has the background knowledge and skills to figure these problems out. They only need to apply their problem solving techniques to the particular facts concerning this distribution system and the problems needing to be solved.

The goal of this project's ES is to assist these users in analyzing normal and alternative system operation configurations. This ES determines which of the system's lines and loads are energized; identifies the paths from sources to loads; evaluates the effect of opening a circuit breaker--what loads and lines become de-energized; evaluates the effect of a faulted circuit breaker or system component--what loads and lines become de-energized and which are rendered unusable; and provides alternative paths for re-energizing de-energized loads.

Having stated the goal of the ES in serving as a distribution system engineering aid, one needs to address why it should. Quite directly, while the current methods of finding answers to these problems work, using an ES is an improvement. The first objection to using an ES might come from the experienced system operator, the human expert, who maintains that everything can already be handled quite nicely without any assistance. Take for instance, however, the problem of identifying what happens when a given breaker is opened. Anyone familiar with distribution systems would consider this problem routine and straight forward. They would have a methodology for determining which lines and loads were de-energized when the breaker opened. They would then need to apply their problem solving method to the particular facts of this problem. The experienced operator may not even need to refer to any drawings for additional information before completing his solution. Of course, in his haste he might overlook something. Others would need more time and reference material. Each would probably reach a solution to the problem. This simple example points out that not all users (or experts) have the same system knowledge and experience or proficiency at problem solving. By encoding the experts' problem reasoning methods and the system information into an ES knowledge base to take advantage of computerized reasoning these types of problems are more quickly solved while ensuring nothing is overlooked and all possible solutions are discovered. Reducing the time required to solve problems and improving the quality of the solution by ensuring its accuracy and completeness answers why this type ES should be used.

In reviewing the needs of the various potential user groups, it becomes apparent that system operators' needs are somewhat different from the other groups. In their case, the SCADA provides on-line sensing of the system's conditions including what

loads are being served. They need an ES that recognizes faults and problems and then determines solutions to the fault problems. The other groups can be characterized as off-line users. They use information that represents what the SCADA provides and want to run 'what if' drills on the system. In this case the ES needs to determine the effects of system operations instead of sensing them as in the on-line case. Since the off-line case eliminates the need to consider the SCADA-DBMS interface and since the off-line users are the target group most likely to embrace an ES as an aid, the off-line needs are examined in this thesis. The ES serves as an off-line engineering aid in analyzing distribution system connectivity problems.

An Expert System and DataBase Management System Approach

This project supplements the ES with a DBMS. The DBMS acts as the source of system specific information and as an information manager. Specifically, the DBMS provides the ES with database information describing the distribution system and the ability to modify that information. This chapter discusses the general features of any relational DBMS. Chapter 2 illustrates specific information management techniques by demonstrating some relational operations using the DBMS written in Prolog to support the ES.

There are two reasons for using a DBMS in this way. First, DBMSs are particularly efficient at storing, manipulating, and retrieving information. Second, efficient management of information argues for a centralized, independent data management system for use by application programs.

The first reason for using a DBMS concerns utilizing its data management strengths. The goal for the DBMS written in this project was to include those DBMS features which most characterize a relational DBMS in so far as they could be reasonably implemented in Prolog and were necessary in the ES companion role. These features include: offering a means of entering information which describes the system; the capability to add to, delete, or change that information; and the ability to take information in one form and change it into another.

In an on-line version of this project's ES, the SCADA's information about line loading, load requirements, and breaker positions would interface to the DBMS. A DBMS routine could take raw data, condition it, and store it as database information. This information would then be available to the ES. For the off-line version of the ES the data is manually entered into the DBMS in its 'natural' form. In addition to storing information which describes the distribution system, the DBMS can be used to describe

modifications to the system. As an example, a contractor can determine how taking several lines out of service affects system operation by deleting the database records that represent those lines and then having the ES evaluate the amended database.

The second reason for using a DBMS is the argument that data should be managed separately from applications that make use of it. From the discussion concerning what distribution system work was done and how people went about doing it, one could see that people applied their own problem solving techniques to the facts of a specific problem. These examples illustrate the idea of separating the application program (problem solving technique) and data itself (the specific problem). These examples support the argument that a problem solving technique is largely independent of the specific information of any given problem.

With this approach the ES serves as the codification of the ways people solve distribution system problems without carrying with it the information about that given distribution system. One practical advantage of keeping system specific data separate from the ES is that the ES then becomes much more mobile. The same ES could be used with other distribution systems without extensive modifications. The DBMS then provides problem specific information and general information management support. For a simple problem with limited information this separation may not seem significant. However, as ESs take on larger systems, the amount of data involved becomes quite significant, so could the advantages gained from using a DBMS.

Expert Systems and Prolog

This section discusses the general nature of Expert Systems (ESs) and implementing them in Prolog. ESs are computer programs which, by reproducing the methods used by human experts, provide assistance in domain specific problem solving. Domain specific problem solving means the ES concentrates on solving problems in a single topical or functional area. This ES's domain is connectivity within an electrical distribution system. Providing assistance may mean faster solutions, being able to consider larger systems or more complex problems, or simply not overlooking any of the problem facts or solutions.

To solve any problem an ES must apply the human expert's problem solving know-how to the facts of the problem. Expert know-how includes formulas, relationships, procedures, and instincts in the form of reasoning rules. These rules and the problem facts are codified to form the ES's knowledge base. The ES's inference engine then applies appropriate rules to the available facts to reach problem solutions.

As an ES is a computer program, the knowledge base and inference engine must be encoded in a computer language. Traditional programming languages handle encoding of facts without any problem. For example, to encode the maximum current capacity of a specific line, programmers can use a variable assignment such as line_current(L114Z)=7000. However, encoding the other half of the knowledge base, the reasoning rules, traditional programming languages do not handle well. The difficulty arises from traditional computer languages emphasizing variable assignment and program control. They specify how variables are updated and used based on other variables using an essentially sequential program control. The programmer concentrates on describing a rigid control sequence of how the program (and variable assignments) will be carried out. What is being calculated though is usually implicit.[6] These languages are described as imperative for they give commands and direction. This programming method does not readily duplicate human reasoning.

A recent class of programming languages is non-imperative or declarative. Their emphasis is not on how the program is executed but rather on describing what is being evaluated or calculated. These declarative languages include LISP, OPS83, and Prolog. Borland Turbo Prolog (Prolog) was selected to implement both the ES and the DBMS. There were two primary reasons for this selection. First, Prolog which stands for Programming in Logic, uses a predicate calculus rather than depending on sequential instructions. The predicates form conditional 'if-then' rules which allow Prolog to deduce or infer new facts from other facts. This ability to infer conclusions from other information is a characteristic of human reasoning. Second, the task of integrating the various knowledge base rules and facts to reach a conclusion is taken care of in Prolog. Prolog does the integration with its built-in inference engine. These features make Prolog a good candidate for modeling this type of human reasoning.

A general discussion of Prolog programming follows. A full treatment of standard Prolog is given by Clocksin and Mellish[7]. Borland Turbo Prolog is a superset of standard Prolog and is described in various publications as well as its manufacturer's manual[8].

Prolog programs are based on an application of predicate calculus. Predicate calculus or propositional logic is built around the use of predicates. Predicates are functions which return either a true or false value when evaluated. A predicate evaluates true only if its argument variables are substantiated or matched with known values. The simplest Prolog program contains a 'predicates' and a 'clauses' sections.

The 'predicates' section simply declares the predicates to be used in the clauses section. A predicate expresses the relationship between the information of its

arguments. It has a name and an ordered set of arguments or attributes. The syntax of a predicate is 'name(a1,a2,a3)'. As an example, the predicate 'bkr_info(name, position, rating)' maintains information concerning a breaker's name, its present position, and its instantaneous current rating.

The program's clauses section consists of fact clauses and rule clauses. Upon program execution these are active in RAM and become part of the program's working memory. A fact clause is merely a single instance of a predicate. For example, for the predicate bkr_info(name, position, rating), bkr_info("SD1A","open",10000) is a single fact clause. Each fact clause can be likened to a database record. Rule clauses take individual predicates and form them into conditional if-then statements.

Rules are used to reach implicit facts or conclusions. In Prolog the form of the rule is 'A IF B' or, stated in 'if then' form, 'IF B THEN A.' The rule can be viewed as two parts--the left hand side, LHS, and the right hand side, RHS. The LHS of the clause acts as the current goal trying to be satisfied. The RHS is a combination of predicates which make up the conditions or subgoals necessary to satisfy the LHS. For example, the following rule determines the position and rating of a breaker.

```
bkr_info(Breaker, Position, Rating) if
                                    bkr_position(Breaker, Position) and
                                    bkr_rating(Breaker, Rating).
```

If the program were trying to determine the position and rating of the breaker SD1A the goal bkr_info("SD1A", Position, Rating) would invoke this rule. For this rule to evaluate true, clause facts must exist or be deduced for bkr_position("SD1A", Position) and bkr_rating("SD1A",Rating). Each of these RHS predicates become subgoals of the rule. The subgoals are satisfied if values are matched or 'bound' to the variables 'Position' and 'Rating.' Values bound to RHS predicate variables also become bound to the same variables in the LHS of the rule. Once the LHS is satisfied by the binding of values for SD1A's rating and position, an implicit fact is established and the rule returns true. If either of the subgoals bkr_position("SD1A", Position) or bkr_rating("SD1A", Rating) are not substantiated, the LHS is not satisfied and so the rule returns false. Hence, Prolog relies on if-then rules to reach conclusions.

The Prolog programmer concentrates on the relationships the rules represent. Consider the case where the method of calculating delivered power is dependent on system configuration. There might be six different configurations to consider. Each system configuration would require its own power calculation formula. In Prolog, the

six configurations and calculation methods would require six rules. In addition to one of the power formulas, the RHS of each rule would include the predicates necessary to uniquely determine the system configuration appropriate for that power calculation formula. Other than including those conditions necessary for using any given rule, the programmer provides no program direction as to when the program goes to or uses any one rule. In execution Prolog handles the details of matching the situation (facts) of each configuration to the power calculation formula. In this way the programmer concentrates on describing how the LHS of the rule is satisfied by the combination of RHS predicates.

The whole set of rules along with any clause facts make up the clauses section. Program execution is begun with a user question (goal) in the form of a predicate. Prolog then selects rules which are appropriate for satisfying the goal or subgoal being pursued.

The implicit facts developed in satisfying a goal or subgoal are transitory in nature. Unless made part of the working memory, the facts developed exist only while attempting to satisfy the subgoal. To support the need to make deduced facts permanent and likewise remove facts no longer true, Prolog supports a dynamic database. This feature allows the program to have permanent rules, permanent facts, and dynamic facts in working memory.

The command 'assert' places facts into the dynamic database. The command 'retract' removes facts from the dynamic database. Predicates for the dynamic database are declared in the 'database' section. These predicates, just like regular predicates, are used as part of rules within the clauses section. To become facts within working memory, however, the program must take positive action and 'assert' a database predicate, with its given set of argument values, into the dynamic database as a fact. When a given fact is no longer needed or is no longer true, the program takes positive action to 'retract' the clause fact from the dynamic database.

The distribution system ES consists of a distribution system knowledge base, Prolog's built-in inference engine, and a querying interface. The distribution system knowledge base consists of if-then rules representing the ways distribution system experts step through solving various problems and predicate clause facts representing distribution system information. A querying interface allows the user to submit questions (goals) which the inference engine attempts to answer using the knowledge base information.

DataBase Management Systems

The inclusion of a DBMS as part of an ES is based on its data management power and convenience as a central data source. Most engineers are comfortable with the concept and use of databases. The DBMS as a program which works with databases is also a comfortable idea. Most, however, have only a fuzzy feeling for the full capabilities of DBMSs. The purpose of this section is to give some clarity to the notion of a DBMS, particularly the relational model DBMS, and, in doing so, to further develop the distribution system model. The convenience of using a DBMS as a data source should also become clear. For a complete treatment of the DBMS topic refer to a text on the subject.[9]

A design engineer may decide he needs some organized method of keeping track of all the possible circuit breaker types used in distribution system design. He could develop a 'distribution_breakers' information table consisting of rows and columns. Each row acts as a record of information for a particular circuit breaker type. Each column stores information about some attribute of the circuit breaker. In this example the following might make up three records from the database:

type breaker	voltage	max current	type relay
1xas2	12.5	100,000	c
1xas3	12.5	250,000	b
2xas3	7.5	75,000	a

His entire table would consist of all the records made up in this form. A database program would include the functions necessary for entering, modifying, and retrieving records from this table.

A DBMS, however, consists of much more. A DBMS is a program composed of those procedures necessary to facilitate database creation; data input, storage, and retrieval; and database manipulation. The power of database manipulation is what most characterizes DBMSs.

The power and flexibility in making database manipulation operations depends on the data model. While data can be modeled in several ways, the relational model is considered by many to best achieve the needs of maximizing flexibility while retaining manipulative power. Most DBMS models treat the relationship among record data similarly, as an ordered set of attributes. In the relational model, the entire set of records (the table) is called a relation. All the relations on a particular subject, the distribution system for example, make up the database. What further differentiates

data models is how they handle connections between relations. In the relational model there is no fixed connection between relations. The lack of any fixed connection between the relations allows the information within the database to be reorganized as desired to form new relations using attributes common between relations.

For an engineer, the relational DBMS serves as follows:

-1) provides information in a user understandable form;

-2) provides information in its most basic and independent form;

-3) maintains database consistency and integrity;

-4) provides informational tools useful in problem resolution.[10]

Information in a User Understandable Form

Effective use of any system requires that the user understand the system's organizational model and be willing to work with that system. So it is with DBMSs. The user must be able to organize his data in an easily understood form. Relational databases organize information in tabular form. As with other database models, each row is a record. A record holds a set of related attribute values. Each column of the table stores a particular attribute of the table. As an illustrative example, consider the relation BRANCH modeling the topology of the Bangor submarine base substation #1 shown in part as figure 2. Table 1 shows the relation for this figure's topology. As with other models the columns represent relational attributes--node field, element field, and via field. Each row is a relation entry. Each entry within the relation is unique.

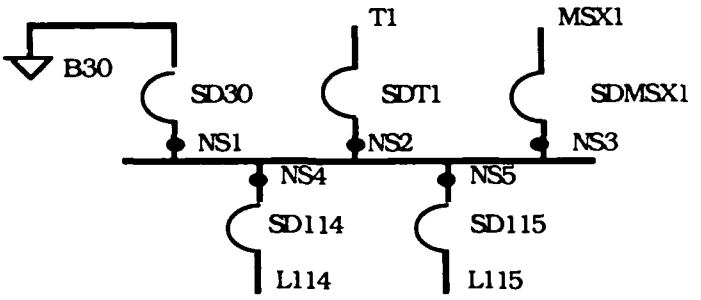


Figure 2. Substation #1, in Part

Table 1. BRANCH Relation for Substation #1

attributes:	node field,	element field,	via field
entry #1	ns1,	sub1,	none
entry #2	ns2,	sub1,	none
entry #3	ns3,	sub1,	none
entry #4	ns4,	sub1,	none
entry #5	ns5,	sub1,	none
entry #6	ns4,	L114,	SD114
entry #7	ns1,	B30,	SD30
entry #8	ns2,	T1,	SDT1
entry #9	ns3,	MSX1,	SDMSX1
entry #10	ns5,	L115,	SD115

Information in its Most Basic and Independent Form

While these relations can be developed in any manner, data independence requires that the relation be based on the data's natural relationships rather than any specific application. If one were developing a database to support a specific application it may seem more reasonable to model the system and data management toward that specific application. That might be an acceptable approach under certain conditions. These conditions would include being sure that the nature of the application will not change and require that the data be changed; that the data being collected for this application is not already available elsewhere or similarly needed elsewhere; and that the results of the application program using this data are not of interest beyond the application program's purpose. Unfortunately these conditions seldom exist. Computer software by its very nature changes. It either does not do exactly what was really required or does it so well that more is desired. Concerning whether the source data is available or needed elsewhere, the very nature of large engineering systems or information systems of any type tend to make information common to several applications. Similarly, when an application program uses information to perform its task it is likely that the program's resultant information may be of use to other systems. Having data in its most basic and independent form enables the data to be efficiently shared between applications as well as allowing for the maintenance of database consistency and integrity (discussed below). The process of obtaining, storing, and manipulating information for use by applications is what DBMSs are all about. The current 'state of the art' for structuring data in its principal independent form is "fourth normal form" (FNF) normalization[11]. Developing distribution system relations begins by describing this system's topology. Figure 3 represents a typical portion of the system. The system's topology is modeled as a set of connected branch elements. The branch element end points are labeled as nodes. Note that loads only

have a single endpoint and that each load branch has a switching device associated with it.

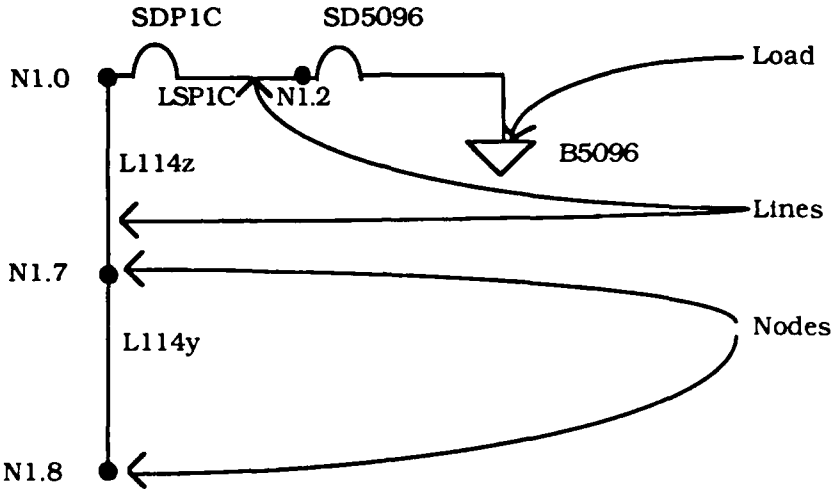


Figure 3. Typical System Portion

Figure 4 shows a typical line branch element pulled out of the system's drawing. In any relation, certain attributes are considered key attributes. The key attribute fields uniquely identify a specific relation entry. To represent the distribution's system topology in FNF, the branch element of figure 4 must be characterized by one or more key attributes so that each unique combination of key attribute values matches up with only a single set of remaining attribute values.

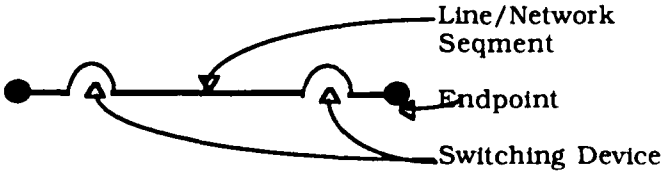


Figure 4. Typical Line Branch Element

Figure 5 shows the previous line segment represented as two 'one end point' segments. When viewed in this way, each of the 'one end point' segments can be uniquely identified by two of its three characteristics-segment name, endpoint name, or switching device name. Segment name and endpoint name are used as the key field attributes which uniquely describe the switching device attribute. The notation for segment name and endpoint name being key field attributes describing the switching

device attribute is:

(endpoint name, field segment name-> switching device).

In the case where there is no switching device the attribute value is filled in by the word 'none'. Table 2 illustrates this relation, BRANCH, describing the topology of figure 3.

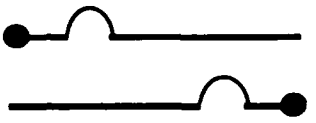


Figure 5. Typical One End Point Line Segments

Table 2. BRANCH Relation for Figure 3. Typical System Portion

attributes:	endpoint name	field segment name	switching device
	N1.2,	B5096,	SD5096
	N1.0,	LSP1C,	none
	N1.2,	LSP1C,	SDP1C
	N1.0,	L114z,	none
	N1.7,	L114z,	none
	N1.7,	L114y,	none
	N1.8,	L114y,	none

As a more complex example, table 3 illustrates this relation describing the topology of substation #1 as shown in figure 2. In this example both key attribute fields are required to uniquely identify various line segments.

Table 3. BRANCH Relation for Figure 6. Substation #1

attributes:	endpoint name	field segment name	switching device
	ns1,	sub1	none
	ns2,	sub1,	none
	ns3,	sub1,	none
	ns4,	sub1,	none
	ns5,	sub1,	none
	ns4,	L114,	SD114
	ns2,	T1,	SDT1
	ns3,	MSX1,	SDMSX
	ns5,	L115,	SD115
	ns1,	B30,	SD30

For the ES two other relations, BDESC and BKRPOS, are needed. BDESC stands for branch description and is described by:

(element -> type, rating).

'Element' refers to the branch element's name. 'Type' refers to the element's type--load, line, or source. The 'rating' attribute serves to store the line's or source's maximum capacity and the load's maximum load value. Table 4 lists the BDESC records for that portion of the system illustrated in figure 2, substation #1.

Table 4. BDESC Relation Records for Substation #1

attributes:	element	type	rating
	sub1,	source,	8E6
	L114,	line,	1E6
	T1,	line,	1E8
	MSX1,	line,	1E8
	L115,	line,	1E6
	B30,	load,	30

BKRPOS represents breaker position. BKRPOS relates each switching device to its current position:

(breaker -> position).

'Breaker' stores the switching device's (breaker's) name. The breaker's position is stored in 'position'. Table 5 lists the BKRPOS records for that portion of the system illustrated in figure 2, substation #1.

Table 5. BKRPOS Relation Records for Substation #1

attributes:	breaker	position
	SD30,	closed
	SDT1,	closed
	SDMSX1,	open
	SD114,	closed
	SD115,	closed

The description of the distribution system is limited to these three relations as they adequately serve to demonstrate the DBMS and contain the information required by the expert system.

Database Consistency and Integrity

The consequence of maintaining data in FNF is that data is in its natural form rather than project specific form. Any given user of the database begins with the same fundamental information. The FNF format allows the DBMS to configure or structure database information as required for use by any specific application. This form avoids unnecessary file duplication and updating problems.

FNF also promotes database consistency and integrity. By assuring that data is maintained in its most basic form, without duplication within the database, every user will get the same data values. Avoiding duplication of data and unnecessary data dependencies also helps control database integrity. Completeness and soundness in the database is the result of a well understood and easily maintained configuration.

Informational Tools for Problem Resolution

DBMSs manipulation strengths provide the engineer with an environment useful for engineering problem resolution. By querying the database, access to required

information is obtained from much larger pools of information. The basic relations can be joined together to create new relations through powerful manipulative capabilities, relational algebra.

Database manipulation consists of both operations within a single relation and operations between two relations. Examples of database manipulation features include: querying the relation based on selection criteria and forming a new relation from an existing single relation or from two existing relations.

As an example of joining two relations together to form a third, consider the following records:

relation BRANCH attributes:	node field,	element field,	via field
	n2.39	bmh8	**sdmh8**
	n2.39	b1014	**sd1014c**
	n2.39	b1015	**sd1015c**

relation BKRPOS attributes:	via field,	position field
	sd1015c	closed
	sd1014c	closed
	sdmh8b	open

relation BRANCHOPEN attributes:	via field,	element field,	position field
	sdmh8b	bmh8	open
	sd1014c	b1015	closed
	sd1015c	b1015	closed

The first two databases act as the source databases. The third database results from joining the first two. As shown, both contain the attribute 'via field' which stores information concerning circuit breakers Those records with common 'via field' values (shown in bold type) serve as data sources for the new database record. The third database contains this common field as well as the 'element field' taken from the first database, and the 'position field' taken from the second database.

In this example the BRANCH and BKRPOS relations were joined together using the common attribute called 'via field' to form a new relation BRANCHOPEN. The new relation contains the database information concerning which element each breaker is associated with and the present position of each breaker. In general, in the relational model DBMS, relations can be joined using any or all of their common attribute fields.

ES Working With a DBMS

This paper proposes that an ES and a DBMS work together as a distribution system engineering aid. The DBMS supplements the ES by acting as its source for information about the distribution system. The ES takes this information and together with its rules solves problems concerning system connectivity.

For the ES to solve these problems it needs to know about: system topology--how the system branch elements are connected together; the function of each system element--does it use power (a load), transmit power (a line), or provide power (a source); and the status of system elements--are circuit breakers and switching devices open or closed. This is the information required from the DBMS.

How does the DBMS get this information? Some of the information, system topology and any element's function, is static. Static information can be manually entered into the DBMS one time as the relations BRANCH and BDESC. The position of system switching devices is not static. The SCADA system can provide this information. In the case of an on-line ES there would need to be a hardware/software interface between the SCADA and the DBMS. This project, however, considers the off-line user. For their purposes there is a set of data for the 'normal position' of all devices. The relation concerning switching device positions is BKRPOS.

These relations comprise the database information for the ES. From this information the ES determines the system's initial conditions. Should the user desire to begin with a non-normal initial configuration, one modifies the original input relations by using the DBMS. Using the DBMS is necessary when testing design alternatives which require modifying existing topology. Alternatively, the user can input the normal configuration and interactively, with the ES, test system response by opening switches or removing lines. The interactive mode is oriented to considering system faults.

The ES requires only part of the information contained in the distribution system database and it requires it in a different form. The ES uses a single relation, BINFO. The format of BINFO is:

(node, branch->type, breaker, position).

For BINFO the FNF is abandoned in preference of a relation structure more suitable for the needs of the ES application. The ES calls upon a relation preparation program to perform the necessary relation manipulations and data format conversion to prepare an ES compatible BINFO relation. Appendix A discusses other general methods of, and obstacles in, linking an ES with a DBMS.

[1] Tomsovic, Liu, Ackerman, and Pope, "An Expert System as a Dispatchers' Aid for the Isolation of Line Section Faults", copyright 1986 IEEE

[2]Komai, Sakaguchi, and Takeda, "Power System Fault Diagnosis with an Expert System Enhanced by the General Problem Solving Method", Mitsubishi Electric Corporation Central Research Laboratory, Hyogo Japan

[3]Liu and Tomsovic, "An Expert System Assisting Decision-Making of Reactive Power/Voltage Control", PICA conference, May 1985, pp.242-248

[4]Talukdar, Cardozo, and Leao, "Toast: The Power System Operator's Assistant", IEEE Computer, July 1986 pp53-60

[5]Liu, S.J.Lee, and Venkata, "An Expert System Operational Aid for Restoration and Loss Reduction of Distribution Systems", Department of Electrical Engineering, University of Washington

[6]Allen and Pokrass, "Logic and Functional Programming", IEEE Potentials, October 1987, pp.21-24

[7]Clocksin and Mellish, "Programming in Prolog", New York: Spring-Verlag, copyright 1984.

[8]Borland International, Inc., "Borland Turbo Prolog version 1.1", copyright 1987.

[9]Tsichritzis and Lochovsky, "Data Base Management Systems", Academic Press, New York, copyright 1977.

[10]Damborg, Ramaswami, Jampala, Venkata, "Application of Relational Database to Computer-Aided-Engineering of Transmission Protection Systems", Energy Group, Department of Electrical Engineering, University of Washington

[11]Fagin, "Multivalued Dependencies and a New Normal Form for Relational Databases," ACM Trans. on Database Systems, Vol. 2, No. 3, Sept. 1977, pp.262-278

Chapter II: The Data Base Management System

DBMS Introduction

This chapter deals with the DBMS developed as part of this thesis project. It is a relational model DBMS written in Prolog. In order to develop further appreciation of the potential of a DBMS as a data manager and engineering tool, this chapter discusses the implemented DBMS features and gives examples of how they perform. Emphasis is also given to the Prolog implementation and program design considerations. Examining the Prolog code concerning its rules and facts provides additional insight into the nature and flexibility of the programming language.

Motivation for Prolog Implementation

The motivation in developing a Prolog implemented relational DBMS was having a DBMS whose structure and code were well understood and available for use as a companion to the ES. This enhanced the joint use of the ES and DBMS by permitting the development of a relation preparation program. The relation preparation program performs the necessary manipulations of the FNF distribution system relations to reorganize the database information into a relation specifically for the ES.

Relation Data Base Management System Overview

The goal for the Prolog DBMS was to included those DBMS features which most characterize a relational DBMS in so far as they could be reasonably implemented in Prolog and were necessary in the ES companion role. The DBMS was developed without further consideration of its role as a ES companion. This approach preserves the basic goal of treating data in its most natural form under the premise that information should be managed on its own merits and not that of the target application.

The basic relational DBMS features included is this project's DBMS are:

friendly user interface;
basic help information;
creating new relations;
selecting an existing relation for manipulation;
entering records into the selected relation;
viewing relations in whole or based on selection criterion;
modifying existing records;
projecting a new relation from an existing one;
joining two relations to form a new relation.

The user interface refers to how the user directs the program to execute commands. A simple DBMS uses menus to lead the user though the necessary steps in specifying commands. While the use of menus is very easy it is also slow and becomes cumbersome as one gains proficiency. More sophisticated DBMSs use a simple natural querying language. Commercial DBMSs integrate these two approaches so that novice users can rely on menus while learning and have benefit of the speedier querying language when the DBMS is mastered. Help information varies in both method of access and degree of detail. The remaining listed functions concern relation manipulation.

These relation manipulation features are selected through the user interface. The DBMS being demonstrated here does not attempt to duplicate all the relational operations found in full featured commercial products. It does, however, try to implement the select, project, and join operations.

In addition to the previously mentioned features the following additional features exist:

directory information concerning available databases;
an erase function for removing database clauses from RAM;
an information window indicating the active relation; and
a time and date display.

User Interface and Command Overview

The user interface adopted attempts a compromise position between the menu and query language approach. A direct query language approach is used for general program direction. It is felt that a user would quickly become familiar with this part of the program's operation and would benefit from the directness of a query language more than the aid of a menu system. Some of the more complex relational operations use a

prompting menu-like approach. A prompting approach recognizes that the user is likely to need more assistance with these operations. Prompting also avoids the necessity of complex query language parsing (decoding of more complex query language commands) or requiring a rigid pattern of simpler direct language commands. This section illustrates the query language by showing its use in invoking the basic program functions. In doing so this section also provides an overview of the implemented program functions. Appendix C contains a further summary of DBMS commands.

The DBMS uses keyword phrases for program direction. These may be of two forms--long and short. An 'action' word and 'object' phrase comprise the first or long form. The action word is the function to be performed by the DBMS. The object phrase indicates the files being operated upon. For example, 'enter r1' directs the program to enter records in the relation r1. The second or short form uses only the action word. For example, 'enter' would direct entering new records into the current relation. Should there not be a current relation, an error message would result requiring a relation to be selected. Some desired actions such as 'join r1 r2 r3' for joining relations r1 and r2 to form r3 have no short form.

The short command 'help' triggers a help information screen in the primary window. This screen provides a concise listing of program functions. Additionally, the key action word accompanies each program function description.

In creating any relation the first action involves defining the relation. The action word 'new' and object phrase 'name' triggers this function. For example, 'new bkrposa' begins creating a relation called BKRPOSA. Note that only single word alphabetic names are supported. The actual defining of the relation involves naming the fields followed by assigning field types and lengths.

At this point the relation BKRPOSA is the active relation. Should the user desire to select a different relation for use, the command 'select' would be used. Should the select command be used alone, without an object phrase, the relation BKRPOSA as the last active relation is automatically assigned as the object phrase. In order to switch to a new relation that relation's name would be used as the object phrase. An example of switching to the relation BRANCH would be 'select branch.'

Once the appropriate relation has been selected the user is ready to manipulate that relation. The manipulation commands involving a single relation include: enter; view; modify; and project. To enter new records the short command is 'enter' while the long command is 'enter branch.' Using the long form of the command avoids the need to use the 'select' command. The program uses the field's name, type, and length as the field input prompts.

In viewing records, the user may choose between viewing all the records within the relation or only selected records. The short forms for this feature are 'view all' and 'view select'. Examples of the long forms are 'view all bkrpos' and 'view select bkrpos'. Again one can switch relations directly by specifying a new relation as the last portion of the object phrase. The 'view all' option lists all records in the order they were entered and stored in the relation datafile. The 'view select' operation is an example of the DBMS using a prompting mode for command input. The 'view select' option first causes each field of the relation definition to be sequentially listed. As each field is listed the user can input the desired matching field criteria or use a '*', the wildcard character, to indicate no specific matching criteria.

The 'modify' command allows for changing existing records. The command's short form is 'modify' and its long form is 'modify branch.' The records to be modified are found based on modification selection criteria. As with the view command, the user indicates which fields have specific selection criteria and which fields have no selection criteria. In a similar manner the user indicates what the new or replacement data is.

Once the basis for determining which records are subject to modification and the nature of the replacement data is complete, the user chooses one of the four modification modes. They are:

global change with no further prompting;

global change with individual record prompting;

delete records with no prompting;

individual record changes.

In the individual record change mode the user may use either the standard replacement data or alter the new data for each record found for modification.

The form of the 'project' command is 'project r1 r2'. This function takes relation r1 and 'projects' it into r2. Projecting involves moving all or selected fields from the original relation into a new relation. Additionally either all records or only records meeting specified selection criteria may be passed to the new relation. The 'project' command automatically takes care of defining the new relation format file. This command has no short form. A detailed example of using the project command to determine all open switching devices is provided later in this chapter.

The 'join' command works on two relations to form a new third relation. It too only has a long form. The command 'join r1 r2 r3' takes relations r1 and r2 to form r3. Like project, all or only selected fields and all or selected records can be passed to the new relation. The key to the join command is specifying fields and selection criteria

which must be common to both the original relations. Using this command simple relations can be combined into more complex relations. For example, the BRANCH and BKRPOS relations could be merged into a new relation 'BRCHOPEN' when needing to determine those branches which contain open switches. This result is possible because the BRANCH relation contains the information concerning the relationship between switches and branches and the BKRPOS relation knows which devices are open. Performing a logical 'and' on these relations creates the desired new relation. Later in this chapter this example is examined in detail.

These data manipulation functions are the primary features of this DBMS.

Coding Overview

This section concerns aspects of implementing a relational DBMS in Borland Turbo Prolog on an IBM AT. Key concepts of a relational DBMS include the relational model and relational manipulation. This section presents the method for storing the relation in RAM and on disk. How the true-false evaluation of predicates is structured to achieve program control and thereby direct relational manipulation is discussed. The last issue presented, relational manipulation methods, concerns trade-offs between *manipulation speed and maximum relation size.*

Relation Structure

A relational DBMS views any relation as a table of ordered attributes. Table 6 shows that part of the relation BRANCH which represents substation #1 as shown in figure 2.

Table 6. FNF Relation BRANCH to Substation #1

endpoint,	field segment	switching device
ns1,	sub1	none
ns2,	sub1,	none
ns3,	sub1,	none
ns4,	sub1,	none
ns5,	sub1,	none
ns4,	L114,	SD114
ns2,	T1,	SDT1
ns3,	MSX1,	SDMSX
ns5,	L115,	SD115
ns1,	B30,	SD30

Prolog recognizes a predicate whose arguments all have specific values, a predicate clause fact, as factual information. For the relation BRANCH the comparative predicate is:

branch(segment_name, node_name, switching_device_name).

Table 7 lists the information of table 6 expressed as Prolog facts.

Table 7. Prolog Clause Facts for Substation #1

branch(endpoint,	field segment,	switching device)
branch("ns1",	"sub1",	"none")
branch("ns2",	"sub1",	"none")
branch("ns3",	"sub1",	"none")
branch("ns4",	"sub1",	"none")
branch("ns5",	"sub1",	"none")
branch("ns4",	"L114",	"SD114")
branch("ns2",	"T1",	"SDT1")
branch("ns3",	"MSX1",	"SDMSX")
branch("ns5",	"L115",	"SD115")
branch("ns1",	"B30",	"SD30")

The prolog predicate 'branch' serves as the means for storing the information in the relation BRANCH. The user sees only the relation's information and not the actual method of storage. The DBMS uses the dynamic database feature to facilitate transferring from the table model of the relation to a Prolog model using predicate clause facts as database records.

For a predicate to be part of the dynamic database it must be declared in the database declaration section. The declaration requirement presents a problem for a general purpose DBMS. For specific Prolog applications the predicate branch(segment_name, node_name, switching_device_name) would be listed within the database declaration section. This approach is unworkable in a general purpose DBMS as the relations are unknown to the programmer. The programmer needs a standard database declaration which will facilitate whatever relations are later defined. The form of the database records is consistent though. As each record consists of a set or list of attribute values and is associated with a specific relation name, the format can be generalized to dbase(relation_name, data_list). This format provides each database record with the Prolog database predicate 'dbase' and two arguments. The first, relation name, links the record data to the correct relation. The second, data list, is a list which contains the attribute values.

Using a list to hold the attribute values solves the problem of being able to support a random number of attributes in each relation. Because Prolog, during program execution, dynamically handles list sizing the number of elements in the list

is unspecified. However, this version of Prolog doesn't support mixed domain lists. As such, each element of the data list must be of the same domain. In the database section of the program code, the predicate is declared as dbase(string, dlist). The argument 'relation name' must be of standard type 'string' and the data list must be of the user defined type 'dlist.' Since this DBMS considers three data types, string; integer; and real, the domain type 'dlist' must accommodate each type. To allow mixed data within the dlist argument, dlist is declared as 'a_field*', a list of type 'a_field.' A_field is defined as s(string) or i(string) or r(string). Dlist represents a list of data functored tokens. The first letter of each functor indicates the data type it represents and the argument the actual information. As the argument to each functor is of type string, the actual information is converted to string for storage purposes. This arrangement handles accommodating the relation as database records within RAM.

Database File Structure

As a superset of standard Prolog this version of Prolog supports storing database information on disk. In storing information on disk Prolog treats each fact clause as a single term. This results in the whole clause being written or read. Table 8 illustrates the same information shown in tables 6 and 7 as 'dbase' predicates. Prolog uses the same format both in RAM and on disk. This format results in a flat ASCII file.

Table 8. BRANCH Database File Sample

```
dbase("BRANCH",[s("ns1        "),s("sub1     "),s("none     ")])
dbase("BRANCH",[s("ns2        "),s("sub1     "),s("none     ")])
dbase("BRANCH",[s("ns3        "),s("sub1     "),s("none     ")])
dbase("BRANCH",[s("ns4        "),s("sub1     "),s("none     ")])
dbase("BRANCH",[s("ns5        "),s("sub1     "),s("none     ")])
dbase("BRANCH",[s("ns4        "),s("L114     "),s("SD114    ")])
dbase("BRANCH",[s("ns2        "),s("T1       "),s("SDT1     ")])
dbase("BRANCH",[s("ns3        "),s("MSX1     "),s("SDMSX1   ")])
dbase("BRANCH",[s("ns5        "),s("L115     "),s("SD115    ")])
dbase("BRANCH",[s("ns1        "),s("B30      "),s("SD30     ")])
```

Program Control

Prolog programs operate by evaluating predicates as true or false. This process of evaluating predicates is referred to as goal satisfaction. Prolog program operation centers around goal satisfaction by trying to find a clause where the predicate evaluates true. Clauses, found in the clause section of the program, can be either rules (LHS IF RHS.) or facts (LHS with no RHS).

For instance:

A Prolog rule:

```
bkr_info(Breaker, Position, Rating) if
                                    bkr_position(Breaker, Position) and
                                    bkr_rating(Breaker, Rating).
```

A Prolog fact:

```
bkr_info(_, _, 'error').
```

A goal is satisfied by either finding a predicate fact clause that matches the goal or satisfying a rule whose LHS is the same as the predicate goal. If the current goal is to satisfy bkr_info("SD114", Position, Rating), Prolog would first try to satisfy the rule shown above by satisfying each of the RHS predicates. Each RHS predicate is a subgoal to the current goal. Should Prolog not be able to satisfy any of the RHS predicates, the rule fails. Prolog does not give up on the goal bkr_info("SD114", Position, Rating) but rather moves to the next rule or predicate fact clause. In this case the clause 'bkr_info(_, _, 'error').' is evaluated. This clause satisfies the goal predicate and returns the string "error" in the last argument position for the variable 'Rating'.

At program execution the program does nothing until it receives a goal to be satisfied. This goal can be thought of as the question being asked. This question can be asked interactively, in the form of an allowed predicate, or can be pre-specified with a clause in the goal section of the given program. The DBMS uses a goal section clause to begin the program's execution. The goal section clause indicates the predicate 'menu' is to be satisfied. The DBMS contains a rule whose LHS is 'menu'. This rule is shown in pseudo-code below:

```
menu if     get user's keyword command phrase input and
            accomplish command task and
            if last command isn't 'quit' and
            menu.
```

The DBMS program then attempts to satisfy the individual predicates which make up the RHS of the rule. Program direction is achieved through these RHS predicate subgoals. One RHS predicate solicits the keyword command phrase input. Then another RHS predicate serving as a subgoal to the menu rule initiates another rule to begin completing the user's direction. There is a continuing sequence of RHS subgoals triggering new rules until whatever command the user gave is completed. Once

the original command subgoal is satisfied the program tries to satisfy the next RHS predicate of the menu rule. The next predicate check if the last command was to quit. Assuming the user doesn't want to quit, the clause recursively calls itself as a subgoal. In this way program execution continues until the user desires to quit. Should the last subgoal have been to quit, the RHS clause fails and the menu clause fails. Likewise the goal clause fails and program execution stops.

Relation Manipulation

Decisions concerning trade offs between speed and maximum database size play a large role in determining methods for database manipulation. In general, disk input/output, i/o, operations reduce execution speed. Not having been able to implement any sophisticated indexing method (such as a B-tree index), bringing the entire database into RAM, performing any necessary operations, and writing the resultant back to disk produces the fastest execution times. With this approach the need to store the original database, support individual record manipulation, and store the operation's results, causes available RAM to limit the database size.

For this DBMS, the project and join commands most relate to this problem. The approach taken here recognizes the importance of execution speed and works around current RAM constraints. Current personal computers running the Micro-Soft Disk Operating System, MSDOS, find themselves limited to addressing 640K of RAM.

The approach taken involves bringing the entire relation into RAM for manipulation. RAM space be used efficiently to allow as much space as possible for storing the existing relation, the result of the relational operation, and to allow for the necessary code overhead (stack space). Of the addressable 640k RAM the MSDOS, program code, and program overhead takes about 240K. 400k of RAM remains for storage of the relation and the results of any relational operation. To make the most of this space during relational manipulation, records from the original relation(s) are removed from RAM once they are no longer needed. Similarly, a new relation's records are written to disk as soon as possible. As such, less space is needed than if both the entire old and new files were co-resident in RAM. In addition to having space available within RAM to store the relation, the stack must be able to handle the overhead associated with the relational operation.

Stack size is limited by the MSDOS to 64k. The stack is used for building structures, parameter storage, and in program calls, subgoal calls, and recursive calls. Recursion involves a rule calling itself. Recursion is used when the number of loops or operations are not known in advance. While the use of recursion is a convenient programming technique, it consumes stack resources until the recursion ends. In

keeping with the basic concept that Prolog evaluates predicates as true or false, the stack resources are released only if a calling predicate (goal) is either satisfied (evaluated true) or fails (evaluated false). The practical result is that all the space available to the stack, 64k, may be consumed before the relational operation is completed. This results in a program error and failure. An alternative method of handling situations where the number of iterations is not known in advance is with the standard predicate 'fail'.' 'Fail' forces program backtracking to the previous subgoal. Additionally, when the program retreats to the previous subgoal stack resources are released. Using the fail predicate in place of recursion allows large relational operations within the 64k stack space limitation.

In summary, current DOS limits RAM to 640k. Of this the DOS, program code, and stack overhead takes about 240k. The remainder, 400k, is available for storage of relations and the results of relational operations. Using the standard predicate 'fail' allows a 64k stack to handle complex relational operations.

Program Operation and Examples

This section demonstrates program operation through the use of three example DBMS operations. These are:

a global 'modify' of the BRANCH relation to eliminate all records without switching devices;

a 'project' action on the BKRPOS relation to create a new relation, BKROPEN, with only open switching devices;

a 'join' operation on the BKRPOS and BRANCH relations to create a new relation, BRCHOPEN, with only those branches having open switching devices.

Examining how these operations are accomplished will include analysis of key rule clauses.

The general approach to any of these DBMS operations is to decipher the keyword phrase command, input additional command directions, read the relation into RAM, perform the directed operation, and write the new file out to disk.

From the 'menu' rule clause the user's input of 'modify branch' is directed to the 'process-modify' rule clause. As can be seen from figure 6, the argument to the process predicate is a string list. The list's head, "modify", matches with this rule clause causing it to be pursued as a subgoal. The relation to be modified, BRANCH, is bound to the variable TC. The first two RHS predicates take the tail of the command and convert it to upper case, the proper format for disk access. After erasing any existing relation

records and formats in RAM, the 'whatis' predicate, by reading the disk based format file, determines the relation format. The format information includes: the relation name (BRANCH); a field length list; a field type list; and a field name list.

Modify

```
process(["modify"|TC]) if
        matchS(TC,Dbl,_),
        upper_lower(Db,Dbl),
        erase_db(_),
        erase_fmt(_),
        whatis(dformat(Db,L,T,F)),
        write("Type in selection criteria:\n"),
        input_fields(dformat(Db,L,T,F),F1),
        write("Type in new data format:\n"),
        input_fields(dformat(Db,L,T,F),F2),
        write("Indicate type modification:\n"),
        write("global with no prompt      -->gn\n"),
        write("global with   prompt       -->gp\n"),
        write("delete with no prompt      -->dn\n"),
        write("individual changes         -->ip\n"),
        readln(C),
        concat(Db,".dba",DN),
        openread(datafile,DN),
        readdevice(datafile),
        mod_read(Db,Pf,F1,F2,C),
        save(DN).
```

Figure 6. Code Listing for 'modify' Rule Clause

The code is now ready for specific modify instructions. The rule prompts the user to indicate, field by field, what the record selection criteria is. That is, how should the DBMS determine which records are to be modified. The rule then asks the user what information should be placed in each field of the selected records. If no change is to be made the wildcard character, *, is used. Four types of field modifications are supported: global change without any prompting; global change with individual prompting before change is effected; global delete without any prompting; and individual record changes.

The first example illustrates a global 'modify' of the BRANCH relation to eliminate all records without switching devices. The user-DBMS interaction to accomplish this operation is shown in figure 7, with user responses in bold type. The user begins the interaction with the command: 'modify branch.' The DBMS indicates the user should provide selection criteria. One at a time the DBMS prompts the user with a field name (along with its type and length). As the values of node and element do not determine record selection the wildcard character '*' was entered by the user. Records representing branch elements without switching devices have the value of 'none' in the 'via' field. The user's responds with 'none' when the DBMS prompts for via

field selection criteria. The record selection responses are combined into a list and bound to the variable, F1.

As the general procedure for the modify operation is to select records and then to replace field information, the DBMS next prompts for replacement field information. This procedure is a program shortcoming in that with the 'delete' option these responses are unnecessary. Any response by the user will work. The new data format responses are also made into a list and then bound to F2. The type of modification to be completed, dn (delete with no prompt), is bound to the variable C.

```
Key Word/Phrase--> modify branch
Type in selection Criteria:
Field Name:node (type= s), MaxLength=8  *
Field Name:element (type= s), MaxLength=10 *
Field Name:via (type=s), MaxLength=10 none
type in new field information:
Field Name:node (type= s), MaxLength=8  *
Field Name:element (type= s), MaxLength=10 *
Field Name:via (type=s), MaxLength=10 *
Indicate type modification:
global with no prompt       -->gn
global with   prompt        -->gp
delete with no prompt       -->dn
individual changes          -->ip
dn
```

Figure 7. Computer Screen 1

Next the general steps of reading the relation into RAM and making the modifications are both completed by the predicate 'mod_read(Db,Pf,F1,F2,C)'. This predicate through its rules acts as a subgoal the to 'process-modify' rule. These rules are:

```
mod_read(Db,Pf,F1,F2,C) if          /*Pr is Present Record*/
        readterm(dbasedom,dBase(Db,Pf)),
        mod_match(Db,Pf,F1,F2,C),
        mod_read(Db,Nf,F1,F2,C).    /*Nr is Next Record */

mod_read(Db,_,_,_,_)  if
        eof(datafile),
        closefile(datafile).
```

The first of these two rules individually reads relation records into RAM from the relation file, calls the predicate 'mod_match' as a subgoal to actually perform the record modification, and then recursively calls itself to continue the process. When there are no longer any new relation records to be read in from the datafile, the 'readterm' predicate will fail causing the first 'mod_read' rule to fail. Prolog will then

go to the second rule and try to satisfy it. This rule checks that indeed there is a end of file condition and then closes the datafile. Once satisfied, the 'mod_read' rule as a subgoal to its calling rule is satisfied and the program returns to the calling rule.

The 'mod_read' rule is an example of not knowing before hand how many iterations will be required. When possible, the 'fail' predicate, because it doesn't create large stack demands, should be used to repeat the rule through backtracking. However, to use this technique the predicates within the rule must be non-deterministic. That is, they must be capable of generating multiple solutions through backtracking. As the 'readterm' predicate is not non-deterministic, recursion must be used in lieu of the fail predicate.

The 'mod_match' rules, figure 8, provide an example of how Prolog determines which rule to invoke. The argument variables to mod_match are: the relation name, Db; the current record being considered, Pf; the selection criteria, F1; the new format criteria, F2; and the type of operation, C.

```
mod_match(Db,Pf,F1,F2,C), if
        not(match(F1,Pf)),
        assertz(dBase(Db,Pf)).

mod_match(_,_,_,_,"dn").

mod_match(Db,Pf,_,F2,"gn") if
        mod_change(Pf,F2,F4),
        assertz(dBase(Db,F4)).

mod_match(Db,Pf,_,F2,"gp") if
        clearwindow,
        mod_change(Pf,F2,F4),
        write("Change top record to bottom? (y/n)\n "),
        write_data(Pf),write_data(F4),
        readdevice(keyboard),readln(C),readdevice(datafile),
        mod_act(Db,C,Pf,F4).

mod_match(Db,Pf,F1,F2,"ip") if
        clearwindow,
        mod_change(Pf,F2,F4),
        write("Change top record to bottom? (y/n)\n "),
        write_data(Pf),write_data(F4),
        readdevice(keyboard),readln(C),readdevice(datafile),
        mod_act(Db,C,||,F4).

mod_match(_,_,_,_,_).
```

Figure 8. Listing for 'mod_match' Rule Clause

Prolog begins with the first rule and checks if it can be satisfied. As the arguments to the first rule are all variables, the passed variables bind to the LHS variables and the

rule will be satisfied if the RHS predicates are satisfied. The 'match' predicate checks whether the record being considered matches the selection criteria. Because of the 'not' modifying this predicate, it will only succeed when they do not match. This condition indicates that the record fails to meet the selection criteria and it is asserted into RAM unmodified. If this version of the rule succeeds, is satisfied, control returns to 'mod_read' for the next relation record. Should the record and the selection criteria match, the 'not' modifier would cause the predicate and likewise this version of the 'mod_match' rule to fail.

When one version of a predicate's rule fails, Prolog moves to the next listed rule. In the second version of these rules, instead of normal variables the first four arguments are the anonymous variable '_' which can be bound (and satisfied) by anything. This is used when the values of these arguments do not matter in the logic of the rule. The last argument of this rule instead of having a variable has the string "dn" shown. To bind to this argument the calling predicate's argument must be the same, i.e. "dn". As the first four arguments contain the anonymous variable, to satisfy the LHS of this rule the calling predicate need only match the last argument with the string "dn". Note that this rule has no RHS. Simply matching the last argument to "dn" satisfies the rule.

The general form of the LHS of the 'mod_match' rules is to pass any necessary information through the first four arguments and the operation to be performed through the last argument. The RHS of the rules performs the necessary operation and asserts the new record into RAM. In the case of the 'delete' operation, the necessary operation is to delete the record, i.e. not assert it into RAM. This rule therefore needs no RHS.

After all the records have been processed the revised relation resides in RAM. Control returns to the 'process-modify' rule where the last RHS predicate writes the relation to disk. The example relation, BRANCH, was a 28,000 character file with 434 records. The DBMS took 5 seconds to read and examine each BRANCH record, identify 193 records containing the value "none" in the via field, assert the other 335 records into RAM, and save the modified relation to disk.

The keyword phrase 'project bkrpos bkropen' activates the 'process-project' rule clause, figure 9. The desired effect of this operation is to generate a new relation, BKROPEN, consisting of all the BKRPOS records which have the value of "open" in the position field. Figure 11 shows the user-DBMS interaction required to direct this operation. Note that the computer prompts the user for 'JOIN' field information. This message is a result of the project command sharing code with the join command.

Project

```
process(["project"|TC]) if
clearwindow,cursor(0,0),
matchS(TC,Db11,TC1),matchS(TC1,DbNew1,_),
upper_lower(DBNew,DbNew1),
upper_lower(Db1,Db11),
wrt_fld_names(Db1),
M=1,join_flds(Db1,M,TN1,TV1),
echo(Db1,TN1,TV1),
positions(Db1,TN1,TP1,TT1,TL1),
DbClauses(Db1,TP1,TV1),
nextstep("n",DbNew,TL1,TT1,TN1),
mkNew(Db1,"",DbNew,_),
write("Project complete.  Hit return to continue."),
readln(_).
```

Figure 9. Listing for 'project' Rule Clause

```
Valid BKRPOS fields:
via  position
Indicate JOIN field #1: via
Indicate selection Value (*=wildcard): *
Indicate JOIN field #2: position
Indicate selection Value (*=wildcard): open
Indicate JOIN field #3:
Selected criteria from dbase BKRPOS:
via *
position open


Project complete.  Hit return to continue.
```

Figure 10. Computer Screen 2

Similar to the 'process-modify' operation, the DBMS prompts the user for responses. In this case the DBMS shows the available (valid) fields for the project operation. The project command allows the user to create a new relation with any or all off the attribute fields of the original relation. The order in which the user gives the field attribute names to the DBMS will be the order the attributes will appear in the new relation. The user also may indicate a selection criteria for each field to be included in the new relation. Figure 10 shows that no selection criteria should be used with the 'via' field values but that only records with 'position' field values of 'open' will be included in the new relation. The DBMS continues to ask for field information until the user responds with only the return key. After the user completes the selection process, the DBMS echo prints the selected fields and the values for each field.

The first new RHS predicate in this rule is the 'positions' predicate. This predicate takes the list of selected fields for the new relation and extracts the necessary

format information: field names; field lengths; and field types from the original relation format file. The rules associated with the predicate 'DbClauses' reads into RAM the entire original relation, finds those records which meet the selection criteria (via field value of "open"), and asserts into RAM the new relation's records.

The relation records which meet the selection criteria need to be 'conditioned' prior to becoming records for the new relation. In the case where not all fields of the original relation are used in the new relation, this conditioning includes extracting the appropriate field information and discarding the undesired field's data. Next the data is reordered as indicated by the ordering the user gave in the criteria selection process. Lastly, before asserting the data as a new record, the data is verified as non-redundant (non-duplicate).

The predicate 'nextstep' creates a format file for the new relation and an empty datafile for the relation. The predicate 'mknew', by writing the new relation to the datafile, completes the 'process-project' rule. The 'mknew' rules illustrate backtracking by use of the fail predicate. The pertinent 'mknew' rules are:

```
mknew(Db1,"",DbNew,_) if
        concat(DbNew,".dba",DName),
        openappend(datafile,DName),
        writedevice(datafile),
        dBase(Db1,Fd1),
        write_terms(dBase(DbNew,Fd1)),
        retract(dBase(Db1,Fd1)),
        fail.

mknew(Db1,"",DbNew,_) if
        closefile(datafile),
        writedevice(screen).
```

The program goes through the first rule until it hits the 'fail'. The 'fail' predicate always fails and causes the program to backtrack to the first non-deterministic predicate seeking an alternative solution from that, the non-deterministic, predicate. In this case, the only non-deterministic predicate is the 'dBase' predicate, the dynamic database predicate. It finds the next relation record. After the dBase predicate succeeds, having found a new relation record the predicates following it are repeated. The process of finding a relation record and writing it to the datafile is repeated until the dBase predicate fails, i.e. there are no more relation records. At this point this mknew rule fails and Prolog moves to the next mknew rule. As can be seen, the next rule closes the datafile and returns output to the screen.

The source relation, BKRPOS, was a 11,500 character file with 250 records. The DBMS took 33 seconds to read in the records, delete 181 records not containing the

value "open" in the via field, create 69 new records in RAM, create the new format file and empty datafile, and write the records to the BKROPEN datafile.

The final example illustrates the 'join' operation. The join operation takes advantage of the DBMS using the relational model. By using a relational model, individual relations can easily be joined to form more complex relations through common attributes. The join operation takes two relations and forms a third. The two original relations must have one or more common attributes. Any number of the common attributes can be used in specifying the join operation. Consider the command 'join branch bkrpos brchopen'. The relations BRANCH and BKRPOS act as source relations. The newly created relation will be BRCHOPEN. This command takes the program to the 'process-join' rule, figure 11. Figure 12 shows the additional specific command direction required to accomplish this operation.

As with the 'process-project' rule, the user is prompted with the first relation's attributes and a request for the first 'JOIN' field. Join fields must be common between the two relations. The selection value specified will determine which records are joined. In this example, records with common 'via' field values will be joined. The user hits the return key when no further 'join' fields are needed.

Join

```
process(["join"|TC]) if
clearwindow, cursor(0,0),
matchS(TC,Db1l,TC1),matchS(TC1,Db2l,TC2),matchS(TC2,DbNewl,_),
upper_lower(DBNew,DbNewl),
upper_lower(Db1,Db1l),
upper_lower(Db2,Db2l),
wrt_fld_names(Db1),
M=1,join_flds(Db1,M,JN1,JV1),nu_flds(NuKey,JN1),nl,
N=1,othr_flds(Db1,N,ON1,OV1),
joinlistS(JN1,ON1,TN1),
joinlistD(JV1,OV1,TV1),
echo(Db1,TN1,TV1),
wrt_fld_names(Db2),
L=1,othr_flds(Db2,L,ON2,OV2),
joinlistS(JN1,ON2,TN2), joinlistD(JV1,OV2,TV2),
echo(Db2,TN2,TV2),
joinlistS(TN1,ON2,TNnew),
joinlistD(TV1,OV2,TVnew),
echo(DbNew,TNnew,TVnew),
positions(Db1,TN1,TP1,TT1,TL1),
positions(Db2,TN2,TP2,TT2,TL2),
DbClauses(Db1,TP1,TV1),write("1st clauses complete"),nl,
DbClauses(Db2,TP2,TV2),write("2nd clauses complete"),nl,
mkfmt(DbNew,NuKey,TN1,TT1,TL1,TN2,TT2,TL2),
mkNew(Db1,Db2,DbNew,NuKey),
write("Join complete. Hit return to continue."),
readln(_).
```

Figure 11. Listing for 'join' Rule Clause

In addition to the information concerning the join field(s), the DBMS needs to know what 'other' fields from the first source relation will contribute to the new relation. Also for the other fields indicated, by specifying specific field values only those records qualifying (meeting the selection criteria) will move to the new relation. From the computer screen, the field 'element' with the wildcard selection value will cause the 'element' field to be included in the new relation without any selection process on the value of the field. The remaining field of the BRANCH relation, node, isn't used.

For the second source relation, BKRPOS, no join field information is required. That is because the join information will be the same for both source relations. The DBMS proceeds directly to soliciting information

```
Valid BRANCH fields:
node element via
Indicate JOIN field #1: via
Indicate selection Value (*=wildcard): *
Indicate JOIN field #2:

Indicate other field #1: element
Indicate selection Value (*=wildcard): *
Indicate other field #2:
Selected criteria from dbase BRANCH:
via *
element *

Valid BKRPOS fields:
via position
Indicate other field #1: position
Indicate selection Value (*=wildcard): open
Indicate other field #2:
Selected criteria from dbase BKRPOS:
via *
position open

Selected criteria from dbase BRCHOPEN:
via *
element *
position open


Join complete.  Hit return to continue.
```

Figure 12. Computer Screen 3

concerning the other fields to be included in the new relation. From the computer screen dialog, the field 'position' is selected with a field value of "open". Again like the 'project' command, the selected criteria for the new relation, BRCHOPEN, is echo printed.

Table 9 shows selected relation records from the two source relations and one resultant record. The first record shows the field names. Shown in bold print is the via

field value, sdmh8, which matches between the shown BRANCH and BKRPOS records. Since the via attribute has been specified as the join field these two records are candidates to be joined together. Since for the 'position' field from the BKRPOS relation was specified as "open" the candidate BKRPOS record must also have the value of "open" in its 'position' field. It does. The last record in the table is the resultant record. The first element of the data list is the via field value of "sdmh8". The second member of the list, "bmh8", is the value of the 'element' field of the BRANCH relation record. The last member of the new record, "open", is taken from the position field of the BKRPOS record.

Table 9. Selected Relation Records

```
dbase("BRANCH",[node field,        element field,    via field        ])
dbase("BRANCH",[s("n2.39        "),s("bmh8        "),s("sdmh8      ")])
dbase("BRANCH",[s("n2.39        "),s("b1014       "),s("sd1014c    ")])
dbase("BRANCH",[s("n2.39        "),s("b1015       "),s("sd1015c    ")])

dbase("BKRPOS",[via field,            position field])
dbase("BKRPOS",[s("sd313        "),s("closed")])
dbase("BKRPOS",[s("sdt31        "),s("closed")])
dbase("BKRPOS",[s("sdmh8b       "),s("open   ")])

dbase("BRCHOPEN",[via field,       element field,    position field])
dbase("BRCHOPEN",[s("sdmh8b "),s("bmh8        "),s("open")
```

The RHS predicates of *the of the 'process-join'* rule are similar to the 'process-project' rule. The DbClauses predicate is used twice. That allows inserting into RAM the candidate records from each of the source relations. The 'mkNew' predicate assumes additional responsibilities within the 'process-join' rule. The 'mkNew' rule matches appropriate candidate records and combines them to form the new record. After forming the new record, it verifies that it isn't a duplicate record, asserts the record into RAM, writes the record to the datafile, and fails in order to repeat the process.

In this example operation, the relation BRANCH was a 28,000 character file with 434 records, the BKRPOS relation was a 11,500 character file with 250 records. The resultant relation, BRCHOPEN, has 4500 characters and 69 records. This join example took 5.75 minutes to complete. This result is definitely not fast. Each of the previous examples where originally run on an IBM AT with a hard disk storing the datafiles. To see how much of the execution time was attributable to disk i/o the project and join examples were repeated with the datafiles stored on a RAM disk. The observed times using a RAM disk were only 1-5 seconds faster than the times observed when the datafiles were stored on a hard disk. The time improvement is so small because hard disks are fast and the DBMS rules limit disk i/o in the project command to two, one read for the source relation and one write to save the new relation, and in the join

command to three, two reads for the two source relations and one write to save the new relation

DBMS Concluding Remarks

This chapter provided an overview of implementing a relational model DBMS in Prolog. This included a brief discussion of DBMS functions, examples of relational operations, and design and programming consideration influenced by the Prolog on an IBM AT environment. From the previous discussions one can see the power of a DBMS as a data management tool. The use of a relational model stated in FNF further enhances a DBMS's capabilities. These features motivate its use as a supplement to an ES

The time required to perform the relational operations is judged as marginal at best. Since the execution times did not improve when using a RAM disk to store the datafiles, the execution time is dependent on the DBMS rule coding. This result reflects the lack of a indexing scheme to rapidly retrieve and match database records.

Chapter III: The Expert System

Expert System Introduction

This chapter demonstrates the ES and validates its use as a distribution system engineering aid. The ES's goal and basis are briefly reviewed. Preliminary comments consist of a brief characterization of the system model and a description of the three classes of problems considered by the ES. This project proposes that the ES and DBMS be jointly used as an off-line engineering aid. For each class of problems, demonstration and validation consists of three principle steps. The first step describes the nature of the given class of problems. Second, examples demonstrate how to present the problems to the ES and how the ES responds. The last step addresses the methodology for solving that class of problems.

ES Goal Statement

The goal of the ES and DBMS package is to provide users with an off-line tool for analyzing problems concerning system connectivity. This goal fulfills the dominant need of those managing the Bangor Submarine Base distribution system which is being able to analyze ways of configuring the system to maintain service to loads.

Connectivity

Connectivity refers to the paths used for transferring power from sources to loads. From the DBMS, three relations model the distribution system. The first, the BRANCH relation, represents the system topology as a set of connected branch elements. Each relation entry consists of a branch node/element pair and a switching device. The node establishes the connection points between branches. The switching device controls power flow through the element. The second relation, BKRPOS, stores switch position information. The switching devices may be either open or closed. The third relation, BDESC, categorizes elements as either sources, loads, or lines. In the natural sense, sources provide power, loads consume power, and lines transmit power. The description 'line' is used for distribution lines, transformers, any other miscellaneous components which do not provide or use power. The ES, using the

information from these relations, examines connectivity, i.e. how power is or could be transferred from sources to loads.

Three general classes of connectivity problems are considered. First, initial system analysis determines the condition of the distribution system based on the distribution system database information. The analysis includes which loads are energized, the branch elements which make up the power flow paths for each loads, and a determination of all energized line elements. The second class of problems concern how the distribution system condition changes as a result of single elements being taken out of service. The elements considered are switching devices as well as the three branch element types--sources, lines, and loads. For loads and lines, taking the element out of service assumes a short circuit fault requiring the opening of switching devices to isolate the fault. For switching devices, removing from service means either a user opening a switch or the switch fails and opens. For these situations the ES determines which loads become de-energized, the closest switching devices to isolate the fault, and what lines become de-energized as a result of isolating the fault For fault isolation, the ES does not consider if switching devices operate automatically or manually. The ES determines which switches must be opened to minimize the affected fault area. Third, the ES determines alternative paths for restoring de-energized loads. Finding a restoration path requires determining routes over currently unenergized lines, but not out-of-service lines, to currently energized lines. The ES does not provide a sequence of switch operations to realize the restoration paths but does indicate what switching devices need to be closed.

Initial System Analysis

Problem Nature. Initial system analysis determines the path(s) energizing each load and identifies the energized distribution lines. To accomplish this, the ES must link with the DBMS to receive the information in the distribution system database. The database consists of three DBMS relations--BRANCH, BDESC, and BKRPOS. The ES uses a single relation, BINFO. BINFO, which stands for 'branch information' has the following form:

(node, branch element->type, switching device, position).

Using the BINFO relation, the 'type' attribute identifies sub-stations and generators as sources. Sources provide power to loads through the distribution lines. To energize a load a continuous path, a loadpath, must exist between a source and the load. To determine how a load is energized the ES starts at the load and investigates all possible

paths leading from it. In this way the initial analysis determines every path, branch by branch, which leads from a load to a source. Normally only one such path exists. As there may be energized branches which are not part of any loadpath determining loadpaths to the loads is not sufficient to determine all the energized branches.

To determine energized distribution lines the ES starts at each source and works outward. Any line connected to the source itself or an energized line is energized. Reaching these conclusions involves applying the ES's rules to the information contained in the distribution system database.

Alternative initial system configurations can also be analyzed. While remaining in the ES, the user can directly enter the DBMS program to change the position of the switching devices to reflect another system configuration. Upon completion of the DBMS operations, ES program execution resumes. The user can then perform an initial system analysis using the modified distribution database. In addition to analyzing the system with different switching device configurations, different line arrangements, i.e. topologies, can be investigated. To determine the effect of removing lines, records are deleted from the DBMS's BRANCH relation. Without entries in the BRANCH relation the lines do not affect the system's topology. Similarly, to determine the effects of new lines, records are added in the BRANCH, BDESC, and BKRPOS relations. Entries must be made in each relation to fully describe the nature of the new lines. Using the DBMS to modify the database relations allows the ES to analyze the current system topology with alternative switching positions or system configurations based on modified topologies.

User/ES Interaction. The ES, like the DBMS, uses natural language querying. The command 'paths' invokes the relation preparation program. The relation preparation program loads into RAM, prepares the relation BINFO, and then the ES resumes. Upon resumption the ES reads into working memory the relation BINFO, and begins determining the loadpaths. The ES responds to the user by indicating which load is being considered. Each loadpath found for that load is displayed. When the ES determines there are no additional loadpaths for the load it finds another load to analyze. This loop continues until all loads have been completed. These loadpaths are asserted as dynamic database facts into working memory.

The command 'energized' starts the ES finding energized lines. Any line connected directly to or through other energized lines to a source is an energized line. As the ES determines a line is energized it displays this conclusion to the user and asserts this new fact into working memory.

Methodology. The methodology for finding all loadpaths involves three steps. First, invoking the relation preparation program and reading into working memory the resulting BINFO database. Second, finding a load not already evaluated. Third, finding that load's loadpath(s). The second and third steps are repeated until no further loads require investigation.

The ES begins by first calling a relation preparation program. This preparation program links to the ES the distribution system database information stored in the DBMS as three relations. They are:

BRANCH: (node, branch element->via),

BDESC: (branch element->type,rating),

BKRPOS: (via->position).

The relation preparation program joins the three relations into a single relation, BINFO, of the form:

(node, branch element->type, switching device, position).

This preparation program is equivalent to the DBMS performing three relational operations--two joins and a project. The first DBMS join operation joins the relations BRANCH and BDESC using the common attribute 'branch element.' As the ES does not need to know about branch element ratings, this attribute is not included in the new relation, TEMP1. TEMP1 has the following form:

TEMP1: (branch element, node->via, type).

The second join operation joins the BKRPOS and TEMP1 relations using the common attribute 'via.' The form of resulting relation, TEMP2, is:

TEMP2: (via, position, node, branch element,type).

While the required information is in a single relation, the ordering of the attributes is not what the ES requires. The DBMS would next have to perform a project to reorder the attributes as required by BINFO. The 'via' attribute of TEMP2 and the 'switching device' attribute are the same attribute with different labels.

In addition to forming a correctly ordered single relation, the DBMS and ES handle the information in a different format. To illustrate, one record of TEMP2 is:

dbase("TEMP2",[s("sdmh8 "),s("open "),s("n2.33 "),s("bmh8 "),s("load ")])

One record of BINFO, in the ES, is:

binfo("n2.33","bhm8","load","sdmh8","open").

The difference arises from the need of the DBMS to use a generalized data storage format whereas the ES, as a specific application, can use a specific predicate format. The link between the DBMS and ES must also perform this format change.

To perform these operations several alternatives existed. The DBMS could be used to perform the two 'joins' and a 'project' to create a BINFO relation in its generalized format. A special format conversion routine could then perform the required conversion. Similarly, the appropriate routines from the DBMS and the conversion routine could be incorporated in the ES itself. This alternative would save having to invoke the DBMS and enter the commands manually. The acceptability of the program size increase would depend on the ability of the remaining RAM to handle the relational operations. A third alternative approach was to perform the format conversion and the relation combinations with routines written for these specific join operations. These routines could also exist as a stand alone program or as part of the ES.

Two of these approaches were examined. The indicated relational operations were performed by the DBMS and a stand alone relation preparation program. The results of the DBMS operations follow:

join branch bdesc temp1:	9.75 minutes;
join bkrpos temp1 temp2:	11.2 minutes;
project temp2 binfo:	3.0 minutes;
total:	23.95 minutes.

These same operations using RAM disk instead of a hard disk were 13 seconds faster. Due to the excessive time required for these three operations, the format conversion routine was not developed. The stand alone BINFO relation preparation program took 1.75 minutes to read the three relations, perform the format conversions, form the new relation, and store the new relation on a hard disk. As neither of these approaches used sophisticated data look-up methods, the time improvement is attributed to using routines for specific relation operations rather than generalized operation routines.

This project implemented the DBMS-ES information link as a stand alone relation preparation program. The preparation routines were not included in the ES program code because one, their inclusion would increase ES code size without offering a significant speed improvement and two, these operations were felt to be more DBMS related than ES related.

Once the relation preparation program prepares the BINFO database, the ES places it in working memory as a dynamic database. The ES next identifies which system elements are loads and then determines their loadpaths. The goal

binfo(Node,Branch,"load",Switch,Position) finds a branch which is a load. Once a load has been identified, the ES traces all connected paths from the load. Any path which ends at a source is a loadpath. The loadpath is then asserted into working memory. Paths which terminate at open switching devices or other loads are not loadpaths. These loadpaths are developed using two rules. Shown in pseudo-code:

rule 1:
Make the current branch the end of the loadpath list if
    the branch is a source.

rule 2:
Add the current branch to the loadpath list if
    its switching device is closed and (1)
    the node/branch pair hasn't been considered and (2)
    there is another branch, branch2, connected to branch (3)
    whose switching device is closed and (4)
    node/branch2 hasn't already been considered and (5)
    it leads to a source. (6)

The first rule recognizes that a source branch completes a loadpath. The second rule builds the list of branches leading to the source. Line (1) recognizes that to be part of the loadpath the branch's switching device must be closed. Line (2) ensures that only paths not previously checked are pursued. The last condition necessary to add this branch to the loadpath list is that it be connected to another branch which leads to a source. Lines (3) through line (6) checks this. Line (6) is a recursive call back to these two rules.

Finding the loadpath for B5730 illustrates the action of these two rules. The part of the system dealing with this search is shown in figure 13. The ES, using these two rules, finds the loadpath(s) for B5730. Rule 1 checks if B5730 is a source. The database clause binfo("n1.10","b5730","load","sd5730","closed") identifies this branch as a load and so the rule fails. The second rule also examines this predicate clause. Since the 'position' attribute indicates the switch to B5730 is closed the ES should follow this path. The closed switching device causes first line to evaluate true. Because this branch has not been previously considered the second line of the right hand side (rhs) evaluates true. Line (3) looks for a branch which is connected to this branch. To be connected to the current branch the new branch must share the node n1.10. From the figure that would be branch L114f. The database clause binfo("n1.10","L114f","line","none","closed") tells the ES that this branch element's switching device is closed. Line (5) is satisfied as this new branch has not been previously considered.

To check if node/branch 'n1.10/L114f' leads to a source the rule recursively calls itself with the current branch becoming 'L114f'. Rule 1 again fails. Rule 2 is now checking branch L114f. This time line (1) finds binfo("n1.8","L114f","line","sdL114f","closed") which is the other node/branch combination sharing the branch L114f. Rule 2 checks that this node/branch pair has not been considered previously. With this combination of rules finding node/branch pairs with closed switching devices the ES can move from node to node and from branch to branch. Line (3) now looks for a branch which is connected to node n1.8. From the drawing, L114y, L114x, and L114g all connect to this node. If the ES selects L114g it will continue the previous process until it runs into a dead end at B5065. The ES backtracks to n1.8 and then may select L114x. The previous process again continues until reaching binfo("n1.12", "Lsp1d","line","sd1m","open"). This branch fails because the switching device is open. The ES again backtracks to n1.8. This time the only remaining untried path is L114y. Depending on how the 'binfo' predicate clauses are arranged in working memory the ES may follow other faulty paths or go directly to sub1 through sdL114. Once sub1 is the branch being considered, binfo("ns3","sub1","source","none","closed") will allow rule 1 to succeed. The list [b5730, L114f, L114y, L114z, sub1] is saved in working memory as loadpath("b5730", [L114f, L114y, L114z, sub1]). The first argument of the loadpath predicate is the load's name. The loadpath list is the second argument to the predicate. After determining one loadpath the ES backtracks through each node investigating all possible paths from that node. For this example, each path eventually leads to an open switch or another load. Therefore B5730 has only a single loadpath.

After all possible routes have been checked, the ES selects another load and reapplies these two rules. This continues until all loads have been worked. Having completed the 'paths' command the ES returns to the user prompt to await the next direction.

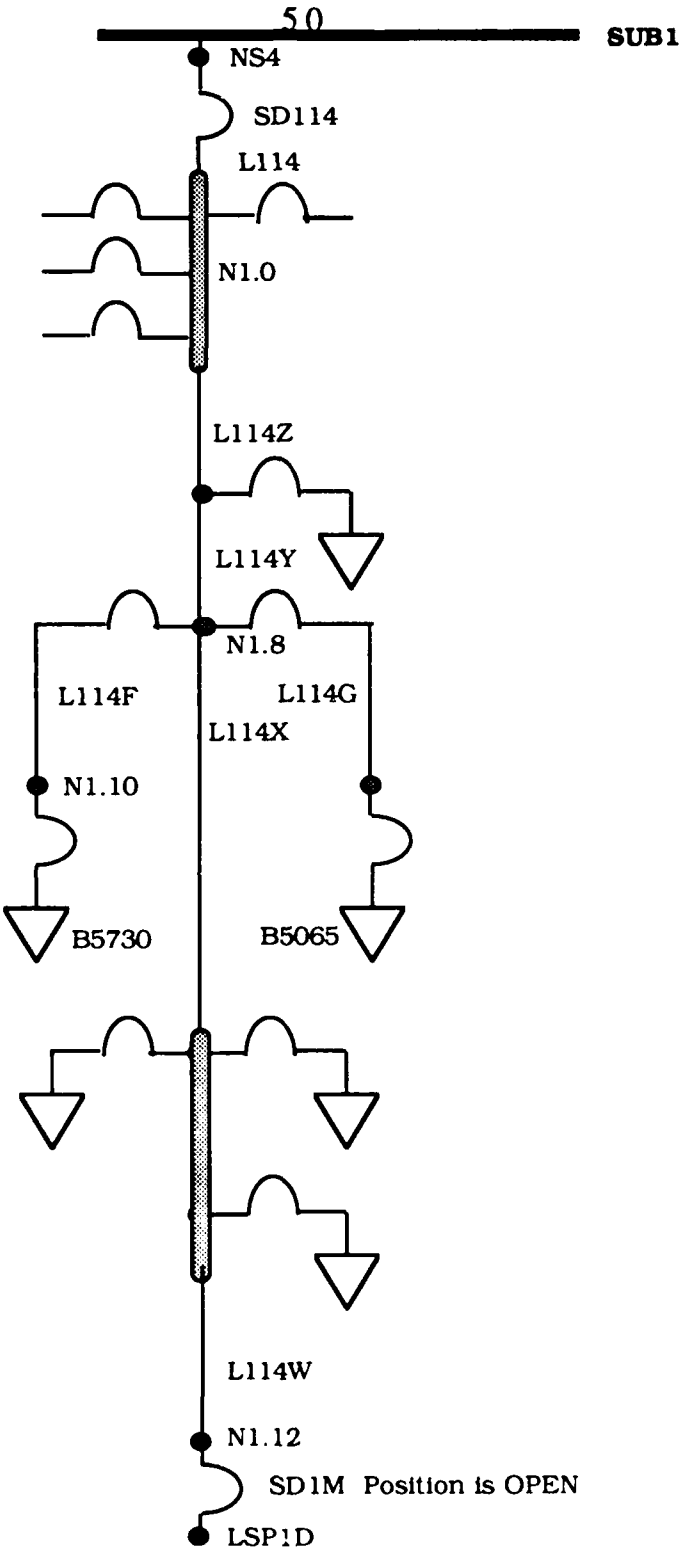


Figure 13. System Extract for B5730 Loadpath

The second part of initial system analysis consists of determining energized distribution lines. Rules similar to the loadpath rules find the energized lines.

rule 3 /* considers branches which are sources*/

node/branch is evaluated if

branch is a source and (1)
node/branch not previously considered and (2)
place in working memory as energized and (3)
find another branch, branch2, to consider which (4)
is connected to this node with a closed switch and (5)
has not previously been considered and (6)
evaluate the node/branch2. (7)

rule 4 /*considers lines with closed breakers at each end*/

node/branch is evaluated if

node/branch not previously considered and (8)
place in working memory as an energized line and (9)
find other node, node2, for branch and (10)
switch for node2/branch is closed and (11)
find another line branch, branch2, to consider which (12)
is connected to this node2 with a closed switch and (13)
has not previously been considered and (14)
evaluate the node2/branch2. (15)

The search for energized branches involves transversing through closed switches the tree made up of interconnected source and line branches. Any such tree begins at a source branch and expands out over line branches until an open switch or a load are encountered. The 'energized' command begins a tree by finding a branch which is a source. For example, the following predicate clauses represent the branch elements which make up sub1:

binfo("ns1", "sub1","source", "none","closed");
binfo("ns2", "sub1", "source", "none","closed");
binfo("ns3", "sub1", "source", "none","closed");
binfo("ns4", "sub1", "source", "none","closed");
binfo("ns5", "sub1"," source", "none","closed").

Rule 3 handles these type clauses. Any one of these clauses can act as the starting point for the ES. Lines 4-6 begin the tree search by finding a previously untraversed connecting branch with a closed switch. Rule 4 line (8) checks that the new node/branch pair have not been previously checked. This rule always evaluates true

when expanding the tree. It is placed as the first rule to promote fast backtracking when the rule fails. Line (9) asserts into working memory that the new branch is energized. Line (10) finds the other node associated with the current branch. Line (11) checks if the switch associated with this node is closed. If it is, the rule looks for another line branch with a closed switch to expand the tree onto. If the switch associated with the second node is open the rule fails. This rule causes the tree to expand along a path until it reaches either a load, a source, or an open switch. The rule then fails and backtracks to an unexplored line branch. When the entire tree is explored rule 2 backtracks to rule 1 and another source branch is found. The new source branch begins a new tree. This process continues until all source branches have been tested.

At this point the ES has completed the initial system analysis. Working memory contains the original BINFO system database, the derived loadpath clauses, and the energized line clauses. The ES is now ready to respond to the effects of opening circuit breakers, single element faults, and restoring de-energized loads. Additionally, the contents of working memory can be saved to disk for later use. The 'save' command saves all database clauses to disk. These can later be read directly into working memory by the 'consult' command. When Prolog saves dynamic database clauses to disk, all clauses of every type are saved in one file. While this project did not do so, a relation preparation program could be developed to convert the individual clauses into separate relations of the general format required by the DBMS.

These two ES operations, finding the loadpaths and the energized distribution lines, are the most computationally intensive. They involve first reading from disk the 441 BINFO relation records into working memory. Every possible path from each load is investigated to see if it leads to a source. From each source every branch is examined to determine if it is an energized line. The process to determine all loadpaths takes 40 seconds to complete. The process of determining energized branches 58 seconds. Several versions of the rules for determining energized paths were examined. While each version of the rule produced the same results their performance varied by 230%. The time improvements resulted from using the fail command instead of recursion when possible and by reordering RHS subgoals. The time differences illustrate that optimized performance results from efficient tree searches and efficient backtracking. The performance and accuracy of the ES in this task cannot be duplicated by even the most proficient user of the distribution system.

Single Element Failure

Problem Nature. The ES handles the distribution system's two most common single element failures. These are a switching device failing open and a branch element shorting. These are the type situations for which the off-line users need the ES to analyze as 'what-if' problems.

The switching device failure results in the switching device being left open without the ability to reclose it. The ES must determine the effects of interrupting power flow through the branch served by that switch. Interrupting power through a branch may cause loss of power to loads and de-energize distribution lines. The ES must also identify the switch as being out of commission (ooc). By noting that the switch is ooc the ES knows the switch is no longer available for use. The ES also accepts an 'open switch' command. The only difference between opening a switch and a switch failing open is that the former isn't labeled as being ooc.

The ES also considers the results of a branch element faulting as a short. Any branch element--a source, load, or line--can be shorted. When a short occurs the ES must determine which circuit breakers open, what lines are de-energized, and which loads are de-energized. In this case the opened circuit breakers are classified as 'lost.' Their use as switching devices are lost because they must remain open to isolate the fault .

User/ES Interaction. Before beginning a 'what-if' simulation, the ES must complete an initial system analysis or have loaded into memory, using the consult command, the results of a previous analysis.

The first of the two types of single element failures is the opening of a switching device. The user gives the command 'open switchname' to indicate simply opening the switch. The command 'ooc switchname' indicates the breaker opens and is out of commission. When opening a switch or breaker the ES first checks the device's current position. If it is already open the user is so notified. If it is closed, its position is changed to open and the user is notified what branch it controlled power flow through. Once the ES has determined which branch had power flow interrupted it can determine the results of the interruption. As the ES identifies de-energized loads the user is notified. Their status is also changed to de-energized loads in working memory. Distribution lines which become de-energized are removed from working memory.

The other type failure is a branch element shorting. When dealing with a shorted branch element the ES must first find the area necessary to isolate the short. The ES determines the boundaries of this area by checking all paths leading from the

fault. For each path the ES finds the closest switching device which can isolate the fault from the remainder of the system. The ES notifies the user which switches need to be opened. The ES knows that, in addition to isolating the fault, opening a switch interrupts power through that switch's branch. By knowing which branches have had power interrupted the ES can determine which loads are lost in isolating the fault. This information is all reported to the user. The final step of fault analysis is redetermining the energized lines using the method previously discussed

At this point working memory contains the original system information in the relation predicate clauses, the derived loadpath clauses, and the energized line clauses, what switch devices are ooc, and what switches have been opened to isolate faults and therefore cannot be reclosed, and which loads have been lost due to the single element fault. Using the 'save testname' command transfers a copy of the working memory's information to disk. This command creates a new datafile on the hard disk. As with initial system conditions saved to disk, this system condition can later be recalled with the command 'consult testname.'

The user may also study the effects of multiple faults. Multiple faults and switch openings can be simulated by a sequence of single element faults and switch openings.

Methodology. The methodology for handling the effects of opening a switch and a switch faulting open are similar. The only difference in these two situations is that the faulted switch needs to be identified in working memory as ooc. This identification is accomplished by asserting the predicate ooc(switch_name) into working memory. The actual opening of a switch produces straight forward results. If it was already open the ES merely notifies the user and returns to the prompt mode. If not, the ES changes its position in working memory by retracting the binfo clause with a 'closed' position value and asserting a binfo clause with an 'open' position value. Next the ES determines what the switch controlled power through or to. Each switching device directly controls power flow in a single system element. As an example of opening a load breaker consider the system portion serving Quarters Group 7 shown in figure 14.

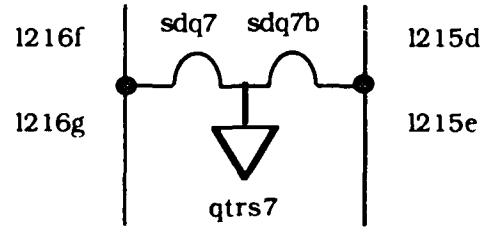


Figure 14. System Portion Serving Quarters Group 7

The user gives the command 'ooc sdq7'. From binfo("n2.20", "qtrs7", load, "sdq7", "closed") the ES knows that switch sdq7 controls power to qtrs7 which is a load. For loads the ES deals only with loadpaths for that specific load. For this load these would be in one of two forms: loadpath("qtrs7", [L216f, ...]) or loadpath("qtrs7", [L215d, ...]). This is because all loadpaths for qtrs7 begin with either line L216f or line L215d. From the figure, one sees that opening sdq7 interrupts power from L216f. Therefore the ES retracts the loadpaths of the form loadpath("qtrs7", [L216f, ...]). The ES next needs to determine if the load, qtrs7, was de-energized. If any loadpaths of the form loadpath("qtrs7", [L215d, ...]) exist the load remains energized. Otherwise the load was de-energized. If the load were de-energized, the ES asserts the inferred fact that the load named qtrs7 was de-energized, deen("qtrs7"), into working memory .

As an example of interrupting power flow through a distribution line consider the command 'open sd114'. This switch is shown in figure 1. This results in the ES finding the clauses binfo("ns4", "L114", "load", "sd114", "closed"). The ES now recognizes that opening sd114 interrupts power flow through the line named L114. To determine what loads may be de-energized the ES needs to determine which loadpaths this line is a member of. Any loadpath with this clause is no longer valid and must be retracted from working memory. The loadpath clauses are of the form loadpath(load_name, loadpath_list). The loadpath_list lists the lines forming that specific loadpath for the given load. The ES retracts those clauses whose loadpath_list contains L114 as a member. Any given load which has had all its loadpath clauses invalidated has been de-energized. To signify de-energized loads the ES asserts clauses of the form deen(loadname).

The ES determines how a branch element fault affects the system in two steps. First it determines the switches which must be opened to isolate the fault. Second it evaluates the effects of opening the switches. The methodology just reviewed for evaluating the effects of opening switches remains unchanged.

To isolate the fault the ES begins by checking if the faulted branch element has a breaker. This initial step is accomplished with the rule:

open the faulted branch's breaker if

assert that branch element as ooc and	(1)
it has a breaker and	(2)
assert the breaker and branch element as lost and	(3)
retract that breaker as open and	(4)
determine de-energized loads and	(5)
continue isolating fault.	(6)

To further isolate a fault the ES must fan outward from the fault along all possible paths. The first switching device along each path becomes the fault boundary for that path. To complete this procedure the ES must recognize all the possible branch combinations and know how to handle each. Figure 15 illustrates the five possible branch situations. While any type branch--source, line, or load--may be considered, the examples will use line branches for simplicity.

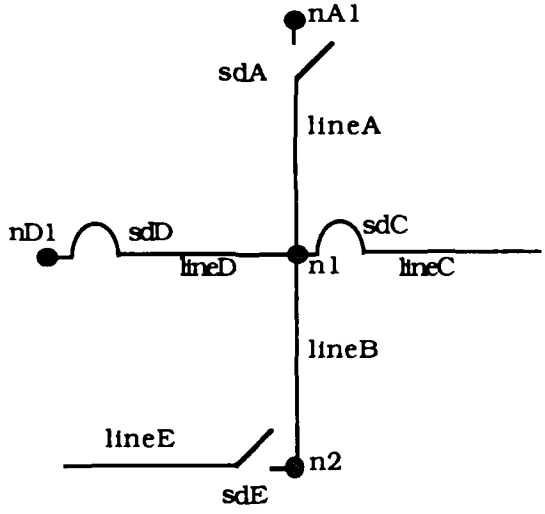


figure 15: Line Segment Types

The binfo clauses of interest for figure 15 follow:

binfo("n1","lineA","none","closed") binfo("nA1","lineA","sdA","open")
binfo("n1","lineB","none","closed") binfo("n2","lineB","none","closed")
binfo("n1","lineC","sdC","closed")
binfo("n1","lineD","none","closed") binfo("nD1","lineD","sdD","closed")
binfo("n2","lineE","sdE","open")

Because the previous rule opens switches adjacent to the fault line, the starting point in isolating any fault will either a type A line, a line with an open switch at one end and no switch at the other end, or type B line, a line without switches at either end. To illustrate the ES actions, assume a short circuit line fault occurs on line A and that sdA was originally closed. The initial rule just discussed opens sdA, asserts into working memory that line A and sdA are lost, and that lineA is OOC. The ES must now move off lineA and determine the remainder of the fault isolation area. This rule is:

given n1 and line A continue isolation search if
there is another branch connected to n2 and (1)
that branch has not been considered before and (2)
using n1 and that branch continue isolation search. (3)

The left hand side (lhs) of the rule establishes the last branch considered. Lines 1 and 2 determine if there is another branch to be searched. Line 3 is a recursive call to cause the search to continue from the old node down the new branch. For this example assume the ES found line D as the next branch. The rule for this type line is:

given n1 and line D continue isolation search if

the branch clause is binfo(n1, line D,_, none,closed) and	(1)
there exists the clause binfo(nd1, line D,_,sdD,closed) and	(2)
sdD represents a switching device and	(3)
open sdD, retract sdD closed and	(4)
assert sdD as lost and	(5)
determine de-energized loads and	(6)
fail.	(7)

This rule recognizes that the end of the branch connected to the search node, n1, doesn't have a switch in line (1). The other end of the branch, at nD1, does have a closed switch (lines 2&3). Line (4) asserts a clause noting that the switch is opened and retracts the clause indicating the switch was closed. Line(7) causes the rule to fail. The fail predicate causes the ES to backtrack to n1 to continue the search. Using the fail predicate conserves stack resources and speeds program execution. The type A rule now must find another path, assume line C. The rule for type C branches is:

given n1 and line C continue isolation search if

the branch clause is binfo(n1, line C,_, sdC,closed) and	(1)
sdC represents a switching device and	(2)
open sdC, retract sdC closed and	(3)
assert sdC as lost and	(4)
determine de-energized loads and	(5)
fail.	(6)

In this rule lines 1 &2 identify the element having a closed switch adjacent to the search node. The switch is opened to isolate the fault. This rule also fails to cause backtracking. The LineA rule next tries line B. The rule for a type B line is:

given n1 and line B continue isolation search if

the branch clause is binfo(n1,line B,_,none,closed) and	(1)
there exists the clause binfo(n2,line B,_,none,closed) and	(2)
assert line is lost and	(3)
there is another branch connected to n2 and	(4)
that branch has not been considered before and	(5)
n2 and that branch continue isolation search.	(6)

This rule moves the search point from one end of the branch to the other. At node n2 there is one remaining branch, line E. Its rule:

given n2 and line E continue isolation search if

the branch clause is binfo(n2,line E,_,sdE,open) and	(1)
sdE represents a switching device and	(2)
assert sdE as lost and	(3)
fail.	(4)

This rule handles the situation of encountering an open switch. The only working memory adjustment needing to be made is, lost(sdE), indicating that switch sdE is lost from service. This clause is inserted into working memory so that the ES knows it cannot later close this switch as part of a possible load recovery path. This rule also fails causing backtracking to check for other branches.

Through backtracking each node is checked for all branches. Checking all branches at every node isolates the fault. When the backtracking is complete each of these rules will have failed and control returns to the calling rule. Prolog then looks for another version of the calling rule which can be evaluated as true. That version updates the distribution line energized list.

Re-energizing De-energized loads

Problem Nature. During the analysis of faults the ES inserts into working memory facts indicating ooc switching devices, lines within a fault isolation area, and loads de-energized from either a single element fault or from the opening of a breaker. To restore power to a de-energized load there must be a path from the load over usable lines and through usable switching devices to an energized line. By examining the system topology and the status of the system lines and switches leading away from each de-energized load the ES can determine potential restoration paths. In analyzing line status the ES checks that the line branch's status isn't 'lost' and if it is 'energized.' 'Lost' lines cannot be included in restoration paths. When the ES reaches an energized line it has completed a restoration path. For switching devices, the ES isn't concerned about their positions but only if they are available for service--not 'lost.' When the open switches are closed the restoration path and the de-energized load become energized.

User/ES Interaction. To direct the ES the user gives the command 'restore.' The ES first finds a de-energized load and tells the user a restoration path is being sought for that load. If a restoration path is found the ES lists the distribution lines that make up the potential loadpath from the load to the energized line. The ES also lists the

switching devices for those lines. After the ES finds all possible paths it will print out the message ' no further paths .' The process repeats itself until all de-energized loads have been checked. When all loads have been checked the ES lists those loads for which there were no restoration path.

Methodology. The ES uses two rules to find restoration paths for de-energized loads. These are:

rule 1: restore with Branch, Path_list, & Switch_list if
current branch is energized and (1)
assert the restorative Path_list and Switch_list and (2)
write the Path_list and Switch_list. (3)

rule 2: restore with Branch, Path_list, & Switch_list if
Branch is not energized and (1)
Node/Branch has not been considered and (2)
Branch and its switching device are not 'lost' and (3)
append its switching device to Switch_list and (4)
there is a Branch2 which connects to Branch and (5)
Node/Branch2 has not been considered and (6)
Branch2 and its switching device are not 'lost' and (7)
Branch2 is not a load and (8)
append Branch2's switch to Switch_list and (9)
append Branch2 to Path_list and (10)
restore with Branch2, Path_list, & Switch_list.. (11)

Rule 1 is the termination rule. When the branch under consideration is an energized branch the path is completed (line 1). Line (2) asserts into working memory the Path_list and Switch_list. Line (3) displays the solution to the user. The solution includes the list of lines and switching devices in the restoration path. All the lines will be unenergized except the last line. The list begins at the load and terminates with the energized branch. To see if any alternative restoration paths exist, the ES must backtrack along the list of unenergized lines making up the first restoration path. This search is done by the ES forcing backtracking through these two rules until no further solutions are found.

Rule 2 does the actual search for candidate lines. When searching to find the first solution only lines (2) through (11) are necessary. However, to force backtracking after having found a solution path, line (1) is necessary. Line (1) forces the search back down the unenergized line list. Lines (5), (6) and (2) jointly select node/branch combinations to examine the system topology leading away from a de-energized load.

During the first pass through this rule the node/branch pair is the de-energized load and the node that connects the load to the rest of the system. Since this is the first time through the rule, line (2) is satisfied. Line (3) checks that the switching device for this node/branch pair can be closed and allow power flow through the branch. The switch is appended to the Switch_list in line (4). Lines (5) and (6) find a connecting branch which hasn't previously been considered. On the first pass through the rule, Branch2 will be the branch which connects to the de-energized load. The found node will be the node in common with the de-energized load. Line (5) finds this node/branch pair. Line (6) verifies that this pair have not been previously considered. Additionally, to add this branch to the restoration list it must not have a 'lost' switching device, itself be 'lost', or be a load. The 'load' test prohibits the ES from using a load as a distribution line. Line (11) is a recursive call with Branch2 as the new line being traveled in search of an energized line. The node is not included as part of the recursive call. By not including the node in the call, line (2) finds the other node associated with Branch2 and so the ES can find the next connecting branch. This process continues until rule one is satisfied, another restoration path is found, or all connecting branches have been checked and both rules fail.

Examples best illustrate this process. Many of the more important loads have two feeders to provide power. Figure 14 shows this arrangement for *Quarters Group 7*. Consider the normal positions of sdq7 and sdq7b as open and closed respectively. Also assume all of the shown lines are energized. If the user were to issue the command 'open sdq7b' the ES would respond with a statement indicating quarters group 7 had been de-energized. If the user were then to issue the 'restore' command the ES would seek all restorative paths for this load. The restorative paths are:
qtrs7 to L216f through sdq7; qtrs7 to L216g through sdq7; qtrs7 to L215d through sdq7b; and qtrs7 to L215e through sdq7b. Had the user first given the command 'ooc sdq7b', switch sdq7b would be in working memory as a 'lost' switch. As such rule 2 would disqualify switch sdq7b from being in the restorative path and only the first two responses would still be solutions.

As a final example consider a modification situation shown in figure 16. Assume that L114x is being taken out of service for some period of time. The distribution system operation procedures require at least two sources of power for every load. To provide for this requirement the job plans call for opening sd114 and sd1m long enough to cut out L114x and tie in temporary feeders to L115. Currently, the loads shown in figure 16 have two sources of power. The normal source is from sub1 through normally closed sd114. The backup source is through normally open sd1m. The test

proposition states that the temporary feeders will assure a similar normal source and backup source for the loads of figure 16 when L114x is taken out of service. The user needs to modify the DBMS distribution system database in such a way as to represent the temporary feeders and to allow simulation of the proposed sequence of operations. The difficult part is simulating the removal of L114x. The 'ooc L114x' command can not be used because it simulates that L114x is shorted and will form a fault isolation area. The alternative methods include taking the line out of the database prior to the ES's initial system analysis or to introduce one or more switches in L114x to interrupt power flow through L114x. The later method is chosen as it allows the simulation to more closely follow the job sequence. From the DBMS with the 'modify' command the user can change the entries in the BRANCH relation from

branch(n1.8,L114x,none) and branch(n1.11,L114x,none) to

branch(n1.8,L114x,sdt3) and branch(n1.11,L114x,sdt4).

In the BKRPOS add entries showing that sdt3 and sdt4 are closed breakers. When the ES's relation preparation program runs the system description database BINFO will show

binfo("n1.8","L114x","line","sdt3","closed") and

binfo("n1.11","L114x","line","sdt4","closed") instead of

binfo("n1.8","L114x","line","none","closed") and

binfo("n1.11","L114x","line","none","closed").

These modifications change the representation of line L114x to include the closed breakers sdt3 and sdt4. The user can then give the commands 'open sdt3' and 'open sdt4' in sequence to simulate taking L114x out of service.

Also the user adds similar information to represent the temporary feeders, with open switches, between L114x and F115. With these changes complete the user can move to the ES. In the ES, the 'paths' command will begin the initial system analysis. Since the L114x switches are closed and the temporary feeder switches are open, the results of the analysis will be the same normal system operating conditions as before the database changes.

Once the initial system analysis has been completed the user is ready to simulate the job plan sequence. The job plan calls for normally open sd1m to be open and then to open sd114. The ES will respond to the command 'open sd1m' with a statement that it is already open. The command 'open sd114' is given next. The ES will change the position of sd114 in working memory and then determine the affect of opening sd114. The determination will result in the loads being listed as de-energized. The command 'ooc L114x' will cause the ES to consider line L114x as having a fault.

This will open sdt3 and sdt4. L114x along with sdt3 and sdt4 will be placed in working memory as lost.

With the command 'restore' the ES begins finding the restorative paths. For each of B1304, B6019, and B6015 the ES finds two possible paths. The first through sd1m to Lsp1d which is energized. The second through the lower temporary feeder, LT2, and switching device SDNT1 adjacent to node nt2 to F115 which is also energized. The remaining loads which were lost when sd114 was opened also have two restorative paths. The first would be their normal loadpaths through sd114. The alternative would be through the upper temporary feeder, LT1, and the switching device SDNT1 adjacent node nt1 to F115.

From these results a system manager could determine that by closing sd114 and sd1m the figure 16 loads would be energized. The temporary feeders, LT1 and LT2, would serve as an alternative power source when switches SDNT1 and SDNT2 were closed.

The previous ES example steps are listed with their execution times in table 10.

Table 10. Example Execution Times

step	**execution time**
relation preparation program	1 minute 40 seconds
paths	40 seconds
energized lines	58 seconds
open sd114	3 seconds
energized lines	58 seconds
ooc l114x	1 second
energized lines	58 seconds
restore	39 seconds

This chapter shows how the ES and DBMS can jointly be used as an effective engineering aid. The ES contains the rules or methods for solving connectivity problems on an electrical distribution system. The DBMS serves as the source for the system specific information. The DBMS also allows the user to modify the nature of the system information by changing both switching positions and system topology.

In addition to being able to correctly solve connectivity problems the ES offers the additional advantage of not overlooking information thereby avoiding missed solutions. The ES is also fast. Initial system analysis of loadpaths and energized branches takes only 40 and 58 seconds respectively.

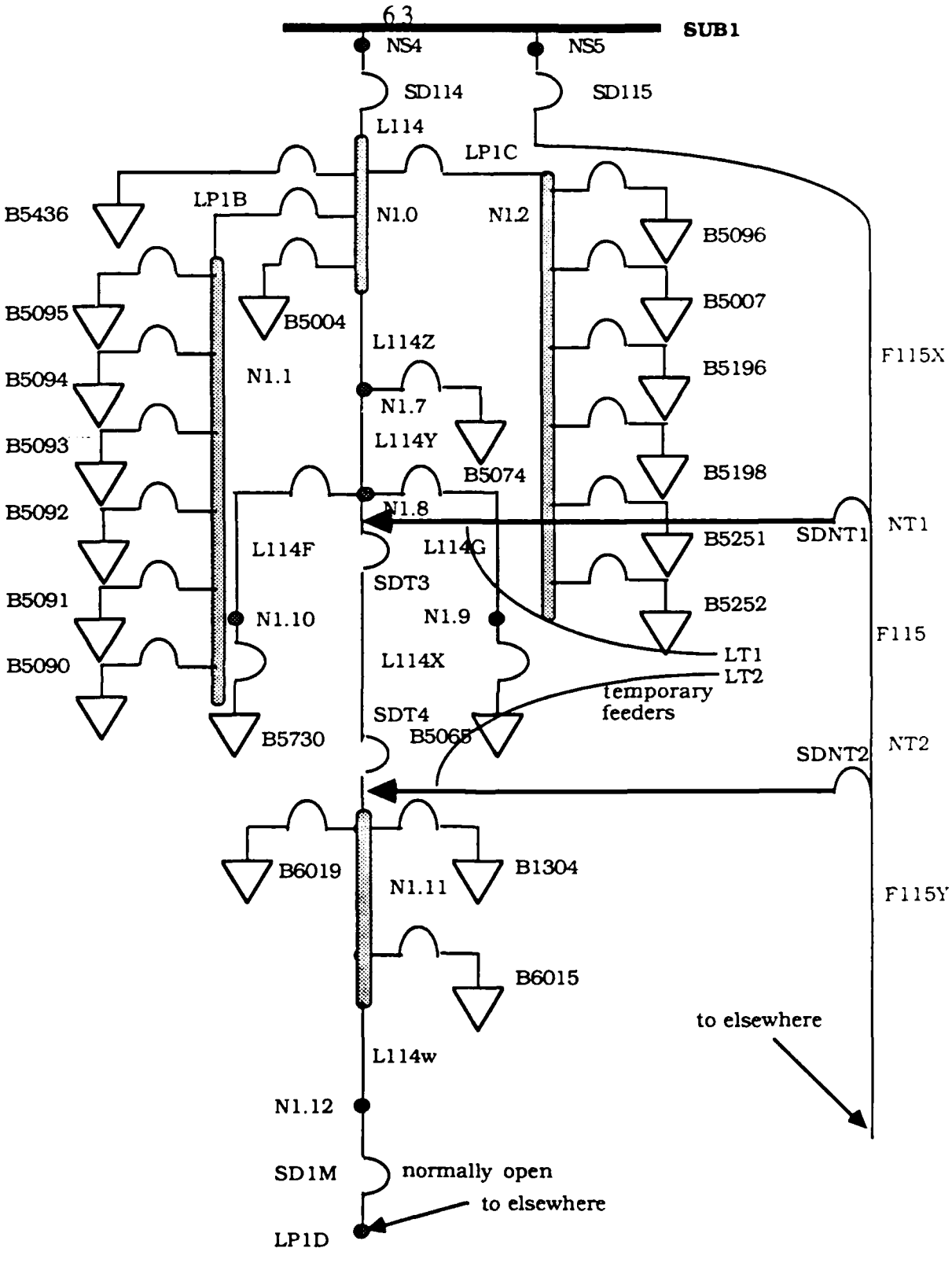


Figure 16. System Modification Portion

Chapter IV: Conclusions

This project has investigated and demonstrated an electrical distribution system engineering aid consisting of an ES supplemented by a DBMS. The DBMS serves as an information manager. It provides for distribution system information collection, storage, manipulation, and retrieval. The ES assists distribution system managers in solving distribution system connectivity problems. The ES contains the reasoning rules for problem solving and links with the DBMS distribution system database for information which specifically describes the Bangor Submarine Base distribution system.

The ES/DBMS approach demonstrates that information can be managed separately from the ES. This approach provides several benefits. By maintaining the information separately the user benefits from the data management strengths of a DBMS discussed in Chapter 2. For the ES user these include: (1) reduced data gathering costs due to central information management and (2) information consistency and integrity as a result of central management using FNF relations. The ES need not bother with as many data manipulation functions. This would take advantage of the DBMS which is written to optimize these type of operations. By shedding the data base management tasks the speed of the ES would improve.

Also, with the ES/DBMS approach the ES stands independently from the information. The same ES can be used at different sites without modification. The consistency of ES code between different sites would ease code upgrade. Using this approach, these traits would hasten the development and improvement of ESs.

Concerning the ES. Human users with a variety of experience currently analyze distribution systems to solve connectivity related problems. The human methods can be encoded using Prolog into conditional 'if-then' rules to allow an ES program reproduce human reasoning. Just as different correct human methods require different amounts of effort to reach a solution, ES rules need to be carefully composed for efficient execution. In part this involves not having redundant subgoals in the RHS of a rule. If the ES has already checked a fact once, checking it unnecessarily a second time slows down execution. Backtracking is a useful tool for finding alternative solutions, however, inefficient backtracking is also slow. Rules which had subgoals added to cause efficient backtracking showed significant execution improvements.

Rule based ESs can easily be expanded and modified. The first version of the ES considered only single branch element line failures. The inclusion of load and source elements failures was completed without the addition of any rules. It was noticed that

generalizing the existing rules automatically accommodated load and source failures. Since Prolog programming emphasizes individual rules the overall program structure was not impacted. By allowing the ES user to move to the DBMS to make database changes without exiting the ES, this project showed that the DBMS can exist in RAM and operate without disrupting the ES.

Concerning the DBMS. The goal in writing a DBMS in Prolog was not to write the ultimate DBMS. The DBMS did illustrate the principles of relational model database management. The performance measurements indicate that indexing schemes such as B-trees are necessary for Prolog coded DBMSs to be acceptably fast.

Linking the ES and DBMS. The relation preparation program used in this thesis demonstrated that the ES can accept information from a DBMS. The approach taken transfers all the information into the ES at once. A better approach would allow for querying the DBMS for specific information and then transferring that information into the ES's working memory. Appendix A discusses other linking approaches .

Limitations and Extensions Each of the three programs developed as part of this thesis--the DBMS, the ES, and the relation preparation program--seemed quite adequate when they were first conceived. After they were developed and used together limitations and desired enhancements became obvious. Some of these, speed enhancements and generalization of the single fault analysis rules, were incorporated into the program code. Others either require more work than time was available for or their implementation was not apparent.

The biggest problem with the DBMS is its performance. Without some indexing scheme the DBMS manipulations are excessively slow.

The ES demonstrates basic connectivity analysis. Currently the substations are viewed as power sources. A more correct view would be to consider the BPA feeder and any active generators as power sources to the substations. This is not a difficult extension as it would require only minor rule modifications. Because the ES views the substations as sources, faults in the 115 kv ring can not be modeled. The ES single fault analysis finds the closest switches which can isolate a fault. An enhancement would also take into account the automatic actions of breakers. This would provide for determining what should be the automatic response of the system as well as the minimum area to isolate a fault. An enhancement to the restoration rules would include further direction as to what switches are currently open and need to be closed and the sequence for closing the switches. As Appendix A discusses, a more generalized link between the ES and DBMS would allow the ES to take advantage of the DBMS information management strengths.

List of References

Allen and Pokrass, "Logic and Functional Programming", IEEE Potentials, October 1987, pp.21-24

Borland International, Inc., "Borland Turbo Prolog version 1.1", copyright 1987.

Clocksin and Mellish, "Programming in Prolog", New York: Spring-Verlag, copyright 1984.

Damborg, Ramaswami, Jampala, Venkata, "Application of Relational Database to Computer-Aided-Engineering of Transmission Protection Systems", Energy Group, Department of Electrical Engineering, University of Washington

Fagin, "Multivalued Dependencies and a New Normal Form for Relational Databases," ACM Trans. on Database Systems, Vol. 2, No. 3, Sept. 1977, pp.262-278

Komai, Sakaguchi, and Takeda, "Power System Fault Diagnosis with an Expert System Enhanced by the General Problem Solving Method", Mitsubishi Electric Corporation Central Research Laboratory, Hyogo Japan

Liu, Lee, and Venkata, "An Expert System Operational Aid for Restoration and Loss Reduction of Distribution Systems", Department of Electrical Engineering, University of Washington

Liu and Tomsovic, "An Expert System Assisting Decision-Making of Reactive Power/Voltage Control", PICA conference, May 1985, pp.242-248

Talukdar, Cardozo, and Leao, "Toast: The Power System Operator's Assistant", IEEE Computer, July 1986 pp53-60

Tomsovic, Liu, Ackerman, and Pope, "An Expert System as a Dispatchers' Aid for the Isolation of Line Section Faults", copyright 1986 IEEE

Tsichritzis and Lochovsky, "Data Base Management Systems", Academic Press, New York, copyright 1977.

Appendix A: Prolog DBMS Links

For the typical ES both the rules and problem facts are part of the program code. The rules and facts establish what the program knows at program execution. The user queries the ES to start its solution mechanism. The ES examines facts and rules drawing inferences, asserting new facts, and retracting facts to reach a solution. When the inference engine is trying to match a proposed fact against the known facts, the program starts at the first clause fact and works its way down the clause fact list until the match is achieved. For large amounts of data this sequential search is relatively slow.

This project maintains that the ES rules and problem information should be maintained separately. Several factors support this position. Beyond the arguments that information should be centrally collected and managed, not all problem facts are static in nature. A DBMS based database easily accommodates the need to change information concerning a problem. ESs are not as efficient as DBMSs for data manipulation. Commercial DBMSs use indexing schemes to reduce data search effort and improve performance. Linking with a DBMS could provide rapid retrieval of information required by the ES. As well as being fast in directly looking up information the DBMS rapidly performs relational operations. An ideal ES/DBMS link would allow the storage of problem information outside RAM. The ES could then access much more information than it could if all problem facts had to always reside in RAM. The ES could also store new inferred facts off line as DBMS relations. There is no question of merit in linking a DBMS to an ES but rather only of methodology.

The relation preparation program used in this project takes advantage of some of these arguments. This approach does not, however, provide a general information linkage between the ES and the DBMS. Brief discussions of more general potential methods of linking ESs and DBMSs together follow. The alternatives considered are:

1) direct access to a DBMS datafiles with data manipulation by the ES;

2) linking to a DBMS with program or command files;

3) incorporating DBMS procedures within the ES.

Accessing the DBMS Datafiles

Accessing of DBMS datafiles means the ES can read the DBMS's information from its datafiles. Use of the DBMS datafile accommodates the issues of expanding the facts available to the ES beyond the RAM capacity and the consideration that data is not always time invariant.

Commercial DBMSs such Rbase 5000 & dBaseIII encode a description of the datafile structure. Properly reading this description reveals the structure of datafile. Both applications store data as a mixed binary/ASCII file. The datafile can be sequentially read and, as desired, the variables bound as ES fact statements. By storing and retrieving data in this manner RAM is used only for the data germane to the current problem solution. This sequentially reading of data while as efficient as the process used by Prolog, fails to take advantage of the more efficient index based retrieval used by DBMS. While the original benefits of non-RAM data storage still exists, execution time is increased as reading information from a disc file is slower than RAM access.

Access to data could be improved if the indexfile could be interpreted. Using the indexfile would allow more direct and faster access to the information. However, as the DBMS manufacturers considered information the indexfile's construction proprietary, its structure is not readily available. Even if the indexfile could be read this approach would probably still be a 'read-only' approach. Without the ability to update the indexfile one could not write information to the DBMS. Also this method does not provide for relational manipulations.

Linking via a File

Commercial DBMSs provide for interactive data manipulation at the prompt level. In this mode the user issues a specific command and the system complies with it. By using a text editor to develop a list of commands the user can develop a program file of commands for the DBMS to execute. It seems reasonable that Prolog could generate a command file such that procedures could be run or more specific searches could be made of datafiles. Prolog does provide a command for invoking DOS commands. So conceptually, Prolog generates a command file then invokes the DBMS to run the file. Prolog allows for the DOS command to invoke the DBMS, the DBMS performs whatever data manipulation is required, an output file is generated, program control returns to the ES, and the ES reads in the file. System performance may be acceptable through the

use of hard disks and RAM disks. This approach remains RAM intensive and does not provide for moving ES inferred facts into the DBMS.

Incorporating DBMS Procedures

Incorporating DBMS procedures in an ES involves taking the code from a full featured DBMS and incorporating whose desired portions into the ES code. This technique is typified by the Program Interface, PI, product for Rbase 5000. The PI code package is a set of DBMS routines which are linkable to MicroSoft Pascal, C, and FORTRAN programs.

With this technique a Pascal, C, or FORTRAN program is written which takes advantage of the DBMS file PI procedures. The PI product once incorporated into the application program no longer requires the Rbase 5000 program code. All that is required is that the database exists. It works directly with the database files.

As Prolog claims it could incorporate procedures written in .OBJ compilable Pascal, C, and FORTRAN, this approach was examined. After achieving only limited success, Borland technical representatives were consulted. They indicated that 'simple' procedure calls were indeed supported. However, as each implementation of higher level languages handled compilation differently, complex calls would not likely work. Borland has announced a fully compatible version of C that was developed with the idea of being able to fully link with Borland Prolog. If a version of the Program Interface were made available that was compatible with Borland C it could be linked to a Prolog based ES. Should DBMS features be incorporated into the ES, most of the advantages of the DBMS would be realized.

Appendix B: System Requirements

The DMBS, ES, and relation preparation programs are all implemented in Borland Turbo Prolog. Borland indicates the minimum system requirements to use Turbo Prolog as:

IBM PC or compatible computer;

384K RAM;

PC-DOS or MS-DOS operating system, version 2.0 or later.

Prolog allows program code to be complied directly to memory while working in the Prolog language. Because of the Prolog system code residing in RAM at the same time the complied program code resides in RAM, 640K RAM and a hard disk are necessary to allow working with reasonably large databases.

However, Borland Prolog also allows the generation of stand alone programs. To run the compiled programs the Borland Turbo Prolog product is not necessary. The DBMS is the largest application developed as part of this project. The DBMS code, the MSDOS, and stack overhead requires approximately 200K. Memory greater than this is available for relation storage and manipulation.

The ES and the DBMS programs as well as all files must reside in the same directory. The user activates either program by typing that program's name.

Appendix C : DBMS Command Summary

select 'relation name'

help

enter ('relation name')

new 'relation name'

view select ('relation name')

view all ('relation name')

modify ('relation name')

- -global change with no prompting
- -global change with prompting
- -delete records, no prompting
- -individual record changes

project 'existing relation name' 'new relation name'

join ' existing relation name' 'existing relation name' 'new relation name'

dir

erase

info

clock

quit

(...) indicates optional command

END

DATE

FILMED

4-88

DTIC